A Milestone of Machine Vision Software

HALCON

HDevelop User's Manual

7.0

A product of MVTEC

HDevelop, the integrated development environment of HALCON, Version 7.0.4

Edition 1	July 1997	
Edition 2	November 1997	
Edition 3	March 1998	(HALCON 5.1)
Edition 4	April 1999	(HALCON 5.2)
Edition 5	October 2000	(HALCON 6.0)
Edition 6	June 2002	(HALCON 6.1)
Edition 6a	December 2002	(HALCON 6.1.1)
Edition 7	December 2003	(HALCON 7.0)
Edition 7a	July 2004	(HALCON 7.0.1)

More information about HALCON can be found at:

`http://www.mvtec.com/halcon/`

About This Manual

This manual is a guide to HDevelop – the integrated development environment for HALCON. HDevelop facilitates rapid prototyping by offering a highly interactive programming environment for designing and testing image analysis programs. Together with the HALCON library, it is a sophisticated image analysis package suitable for product development, research, and education. HALCON provides operators covering a wide range of applications: Factory automation, quality control, remote sensing, aerial image interpretation, medical image analysis, and surveillance tasks.

This manual provides all necessary information to understand HDevelop's basic philosophy and to use HDevelop.

This manual is intended for all new users of HALCON. It does not assume that you are an expert in image processing. Regardless of your skills, it is quite easy to work with HDevelop. Nevertheless, it is helpful to have an idea about the functionality of *graphical user interfaces (GUI)*[1], and about some basic image processing aspects.

The manual is divided into the following chapters:

- **Introducing HDevelop**
  This chapter explains the basic concepts of HDevelop and contains a first example that illustrates how to work with HDevelop.
- **Graphical User Interface**
  This chapter explains the graphical user interface of HDevelop and how to interact with it.
- **Language**
  This chapter explains syntax and semantics of the language used in HDevelop programs.
- **Code Generation**
  This chapter explains the export of a HDevelop program to C, C++, or Visual Basic.
- **Program Examples**
  This chapter contains example programs for typical image processing tasks.
- **Tips & Tricks**
  This chapter explains how to start HDevelop and describes keycodes, warning and error windows, and restrictions.

[1] Consult your platform's documentation for general information.

Contents

Chapter 1

Introducing HDevelop

In fact, HDevelop is more than a graphical user interface to HALCON: It is a highly interactive integrated development environment (IDE) for machine vision applications.

There are three basic ways to develop machine vision applications using HDevelop:

- *Rapid prototyping in the integrated development environment HDevelop.*
  You can use HDevelop to find the optimal operators or parameters to solve your machine vision task, and then build the application using the programming languages C, C++, or COM (Visual Basic, C#, Delphi).
- *Development of an application that runs within HDevelop.*
  Using HDevelop, you can also develop a complete machine vision application and run it within the HDevelop environment.
- *Export of an application as C, C++, or COM source code.*
  Finally, you can export an application developed in HDevelop as C, C++ or Visual Basic source code. This program can then be compiled and linked with the HALCON library so that it runs as a stand-alone (console) application. Of course, you can also extend the generated code or integrate it into existing software.

Let's start with some facts describing the main characteristics of HDevelop, followed by an example session in section 1.3.

1.1 Facts about HDevelop

While developing programs, HDevelop actively supports the user in different ways:

- ☐ With the graphical user interface of HDevelop operators and iconic objects can be directly selected, analyzed, and changed within one environment.
- ☐ HDevelop suggests operators for specific tasks. In addition, a thematically structured operator list helps you to find an appropriate operator quickly.

- An integrated online help contains information about each HALCON operator, such as a detailed description of the functionality, typical successor and predecessor operators, complexity of the operator, error handling, and examples of application. The online help is based on an internet browser such as Netscape Navigator or Microsoft Internet Explorer.
- HDevelop comprises a program interpreter with edit and debug functions. It supports standard programming features, such as procedures, loops, or conditions. Parameters can be changed even while the program is running.
- HDevelop immediately displays the results of operations. You can try different operators and/or parameters, and immediately see the effect on the screen. Moreover, you can preview the results of an operator without changing the program.
- Several graphical tools allow to examine iconic and control data online. For example, you can extract shape and gray value features by simply clicking onto the objects in the graphics window, or inspect the histogram of an image interactively and apply real-time segmentation to select parameters.
- Variables with an automatic garbage collection are used to manage iconic objects or control values.

1.2 HDevelop Procedures

HDevelop offers a mechanism for the creation and execution of procedures. Procedures are meant to increase the readability and modularity of HDevelop programs by encapsulating functionality of multiple operator calls in one or more procedure calls. It also makes it easier to reuse program code in other HDevelop programs by storing repeatedly used functionality in separate procedures.

A HDevelop procedure consists of an interface and a program body. Procedure interfaces resemble the interfaces of HALCON operators, i.e. they contain parameter lists for iconic and control input and output parameters. A procedure body contains a list of operator and procedure calls.

Every HDevelop program is made up of one or more procedures. It always contains the main procedure, which has a special status inside the program, because it is always the top-most procedure in the calling hierarchy and cannot be deleted from the program.

HDevelop offers all necessary mechanisms for creating, loading, deleting, copying, modifying, saving, and exporting procedures. Once a procedure is created, it can basically be used like an operator: Calls to the procedure can be added to any program body and be executed with the appropriate calling parameters. Generally, the concept of using procedures inside HDevelop is an extension to the concept of calling HALCON operators since procedure and operator interfaces have the same parameter categories and the same rules apply for passing calling parameters.

1.3 Example Session

To get a first impression how to use HDevelop, you may have a look at the following example session. Every important step during the image processing session is explained in detail. Thus, having read this chapter thoroughly, you will understand the main HALCON ideas and concepts. Furthermore, you will

learn the main aspects of HDevelop's graphical user interface (for more details see chapter 2 on page 11).

In this example, the task is to detect circular marks attached to a person's body in a gray value image. The program can be found in the file

```
%HALCONROOT%\examples\HDevelop\Manuals\HDevelop\marks.dev
```

You start HDevelop under Windows NT/2000/XP by calling

```
Start ▷ Programs ▷ MVTec HALCON ▷ HDevelop
```

Under UNIX, HDevelop is started from the shell like any other program[1]. Optionally, an application name can be specified as a parameter:

```
hdevelop <File>.dev
```

This application is then loaded. This is identical to an invocation of HDevelop without any parameter and a subsequent loading of the application. If you want to run the application immediately after it has been loaded, invoke HDevelop as follows:

```
hdevelop -run <File>.dev
```

This is equivalent to starting HDevelop, loading the application, and then selecting `Execution` ▷ `Run` in the menu bar of HDevelop.

After starting HDevelop, your first step is to load the image `marks.tif` from the directory `%HALCONROOT%\images`. You may perform this step in three different ways:

- First, you may specify the operator name `read_image` in the operator window's input text field.
- Secondly, you may select this operator via the menu item `Operators` ▷ `File` ▷ `Images`.
- The most often used and most convenient way is the third one. Here, you open the image selection box pressing menu item `File` ▷ `Read Image`. This menu contains several predefined directories, one of which is `%HALCONROOT%\images`. Usually, this directory will be `C:\Program Files\MVTec\Halcon\images`. Select this directory by pressing the appropriate menu button. Now you can browse to your target directory and choose a file name. By clicking the button `Open`, a dialog window appears, in which you may specify a (new) name for the iconic variable which contains the image you are about to load. The variable will be used later in the program to access this image.

  To facilitate the specification process, HDevelop offers you a default variable name, which is derived from the image's file name. Pressing the button `Ok` transfers the operator into the program window and inserts a first program line, similar to the following line, into your program:

  ```
  read_image (Marks,'C:\\Program Files\\MVTec\\Halcon\\images\\marks.tif')
  ```

  This new program line is executed immediately and the loaded image is displayed in the active graphics window. Please note the double backslashes, which are necessary since a single backslash is used to quote special characters (see table 3.2 on page 83). In our example we change the default for the name from `Marks` to `Christof`.

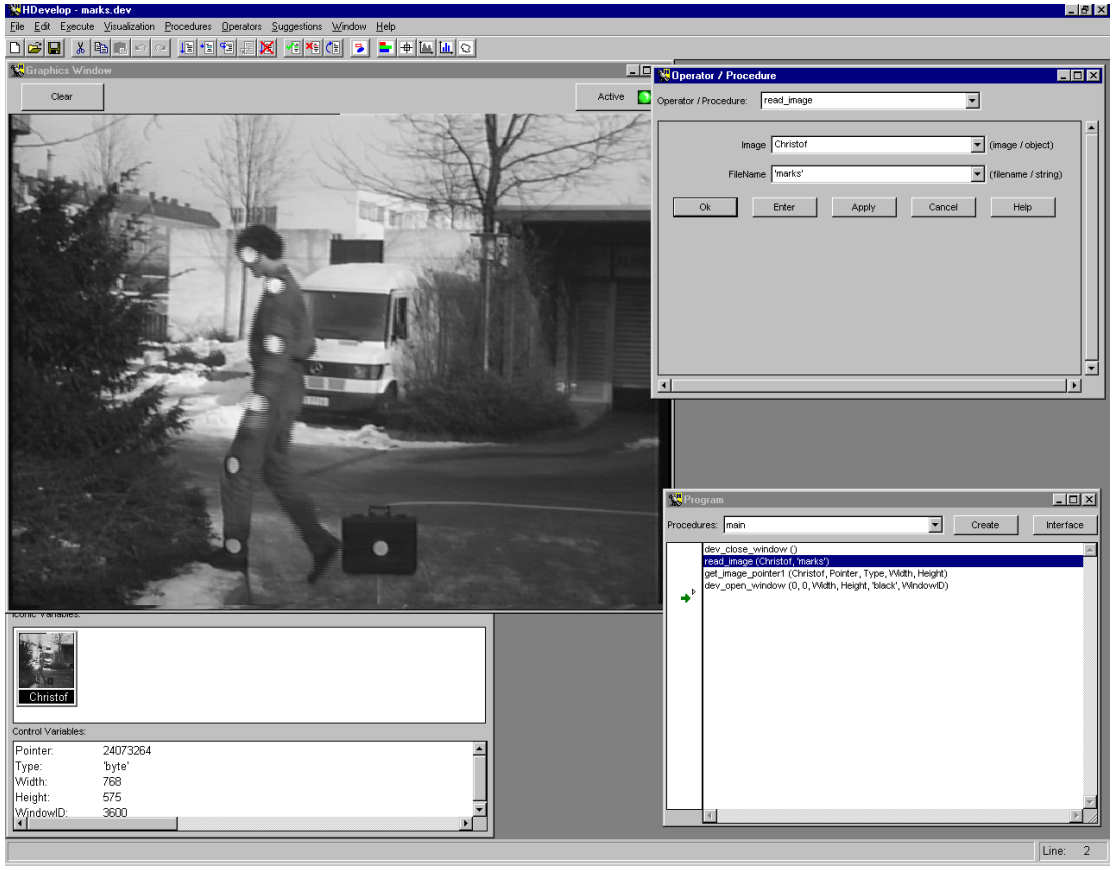

Figure 1.1: Screen configuration after image loading.

Using this selection box, you are able to search images rapidly without knowing their exact file names. In contrast to the two other possibilities, the parameters of operator `read_image` are specified automatically. Thus, an explicit input of path and file name is not necessary in this case. An icon with an appropriate variable name is created in the iconic variable area of the variable window. Double-clicking on such an icon displays its contents in the currently active graphics window. Figure 1.1 shows a complete configuration of HDevelop for the explained scenario. In addition, a new window is opened — after closing the default window — to display the image in its original size.

If you take a closer look at the image in figure 1.1, you will see the typical temporal offset between two half images that occurs when taking images with a video camera. This temporal offset is 20 ms for PAL systems and 16.6 ms for NTSC systems. HALCON offers an algorithm that computes an interpolated full image from such a video image. The name of the appropriate operator is `fill_interlace` (see the HALCON Reference Manual). The next step is to specify this name in the operator window's operator name field. If it is indicated completely, HDevelop shows the operator immediately in the operator window. Now you have to specify the variable name of your image. For this you put in the name

[1] The necessary settings for the operating system are described in the Installation Guide, section A.2 on page 46.

Christof in the parameter field ImageCamera. To do so you have two possibilities:

- Direct input via the keyboard.
- Using the combo box that is associated with the parameter text field, you may choose an appropriate name.

The system's suggestion for the interpolated image is ImageFilled. By clicking button OK you insert this operator into the program and execute it immediately. The computed image is displayed automatically in the active graphics window.

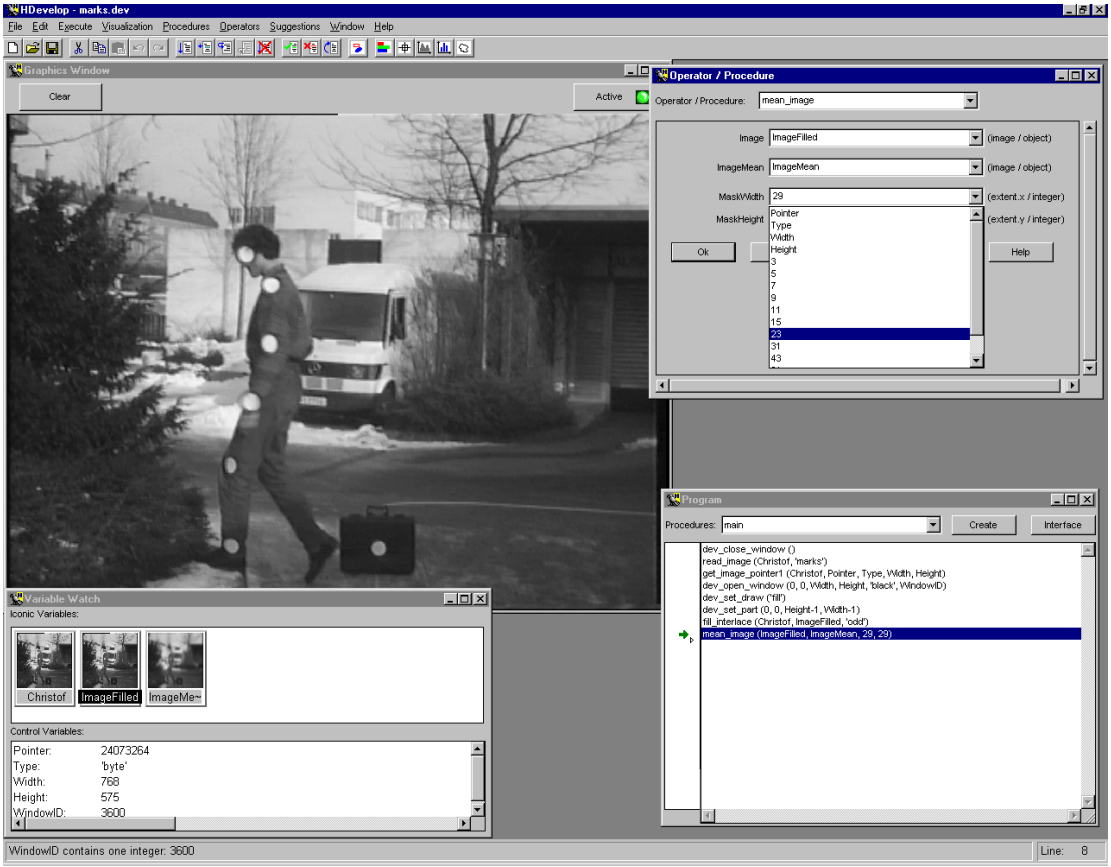

Figure 1.2: With the help of the opened combo box you may specify a reasonable filter size for the operator mean_image.

In the next step you try to separate bright from dark pixels in the image using a thresholding operation. In this case, a segmentation using the simple thresholding operator threshold does not result in a satisfying output. Hence you have to use the dynamic thresholding operator dyn_threshold. For execution you need the original image (i.e., the interpolated full image) and an image to compare (containing the thresholds). You obtain this image by using the smoothing filter, e.g., mean_image. As input image you choose your original image ImageFilled. After estimating the marks' size in pixels, you specify a filter

size which is approximately twice the marks' size (compare the HALCON Reference Manual entry for dyn_threshold).

To choose the operator mean_image, you traverse the menu hierarchy to Operators ▷ Filter ▷ Smoothing. It will be displayed in the operator window immediately. Now you specify the image variable names ImageFilled in the text field called Image and ImageMean in the output text field. The filter matrix size is chosen by opening the combo boxes of the corresponding text fields (MaskWidth, MaskHeight). These combo boxes contain a selection of reasonable input values which is offered by HDevelop. In our example the size is set to 29 (see figure 1.2).

By clicking the button OK you insert the operator mean_image in the program and execute it. Now you have to search for the name of the dynamic thresholding. For this you specify a substring that is included in the operator name in the operator window's operator name text field. Three letters are already sufficient to produce a result. You will notice the open combo box that presents all HALCON and/or HDevelop operators containing the first three specified letters. Now you are able to select the operator dyn_threshold and to specify its input parameters. The value ImageFilled is used for OriginalImage. ImageMean is used as the component to compare (here ThresholdImage). For the output parameter RegionDynThresh the variable name remains unchanged (see figure 1.3).

Image pixels touching each other and remaining above the given threshold have to be merged to single regions. The computation of these connected components is realized by operator connection (menu item Operators ▷ Regions ▷ Transformations). The input region RegionDynThresh is specified in the text field Region. The output variable's default name ConnectedRegions is changed to ConnectedRegionsDynThresh. After the operator's execution all resulting regions are stored in this output variable. This shows a great advantage of HALCON's tuple philosophy: although you have several different results you do not have to worry how to handle them. This is done transparently by HALCON. HALCON operators recognize a tuple type variable and process it accordingly. This results in more compact programs because you may combine several similar operator calls in one operator.

To obtain a better visualization of the results after calling connection you select the menu Visualization ▷ Colored. Here you specify the 12 predefined color presentation. Now every computed region receives one of the 12 colors. This presentation mode is very useful to indicate each region with a different color. If there are more than 12 regions the system uses the same color for several different regions. With dev_display of the image ImageFilled you refresh the graphics window to see the results of next step much better. Select the operator dev_display from the menu Operators ▷ Develop.

In the next step we want to select the regions which correspond to the circular marks of the indicated person in shape and size. This can be achieved with the operator select_shape in the menu Operators ▷ Regions ▷ Features. This operator lets you specify a set of conditions, which the regions have to fulfill in order to be selected. In our case, the circular marks can be selected by specifying two conditions: First, their size is about $15 \times 15 = 225$ pixels. Secondly, they have a circular shape, which corresponds to a compactness close to 1.0 (see the operator compactness for more information about this feature)

Thus, after choosing select_shape you specify the parameters as depicted in figure 1.4:

1. The input region is ConnectedRegionsDynThresh.
2. The output variable name is SelectedRegionsDynThresh.
3. The features are specified as the tuple 'area', 'compactness'.

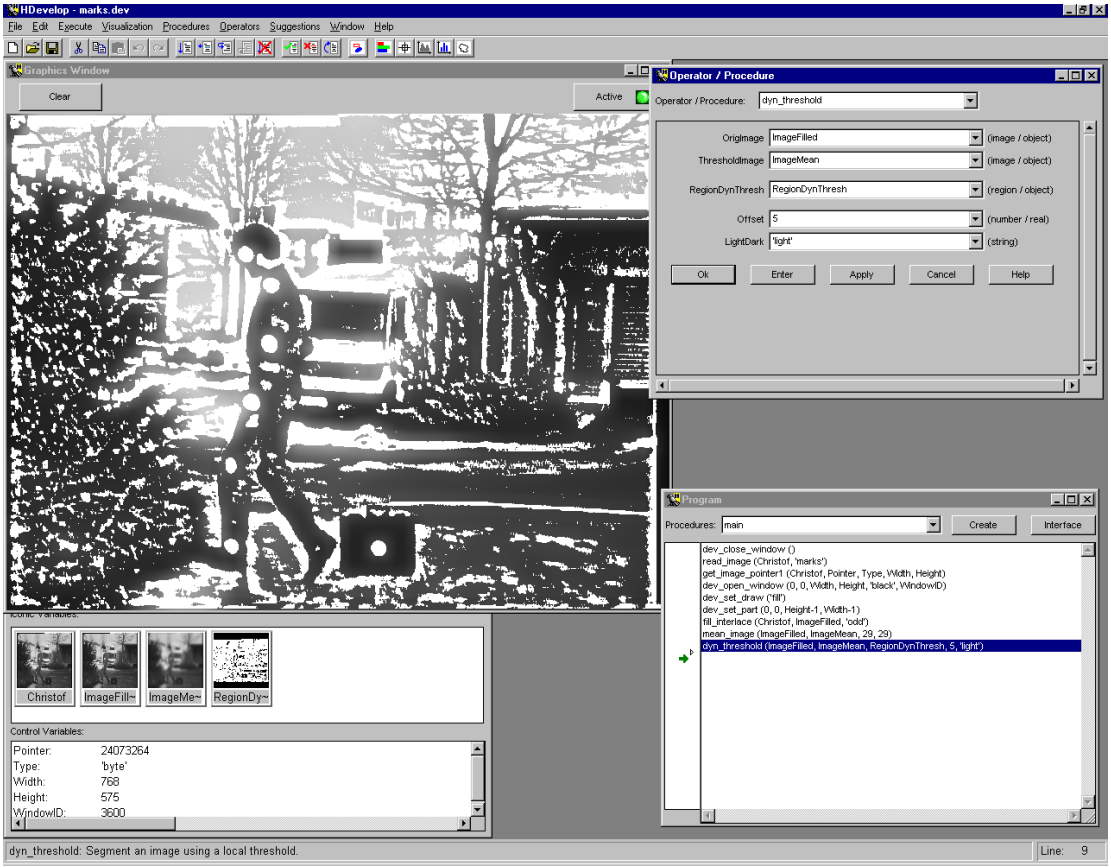

Figure 1.3: The displayed image is the threshold operation result.

4. The value for the operation (`'and'`) remains unchanged, as both conditions must be satisfied in order for a region to be selected.

5. The region's minimum size should be 150, its minimum compactness 1.0 (`Min`).

6. The region's size should not exceed 500, its maximum compactness should be 1.4 (`Max`).

As you can see in figure 1.4, all circular marks are now extracted correctly; however, one additional region is selected as well. Thus, we add a selection of regions based on *gray value* features using the operator `select_gray`. Here, the mean gray value is used to discriminate the objects; it must lie between 120 and 255 in order for a region to be selected. excent shows the effect: Now, only the circular marks are selected.

Finally, we want to obtain some numerical information about the matched marks. For example, we might want to compute three shape features of the marks. They are derived from the regions' geometric moments. The calculation is done by the operator `eccentricity`. The input parameters are all regions of the variable `Marks`. The computed values `Anisometry`, `Bulkiness`, and `StructureFactor` are

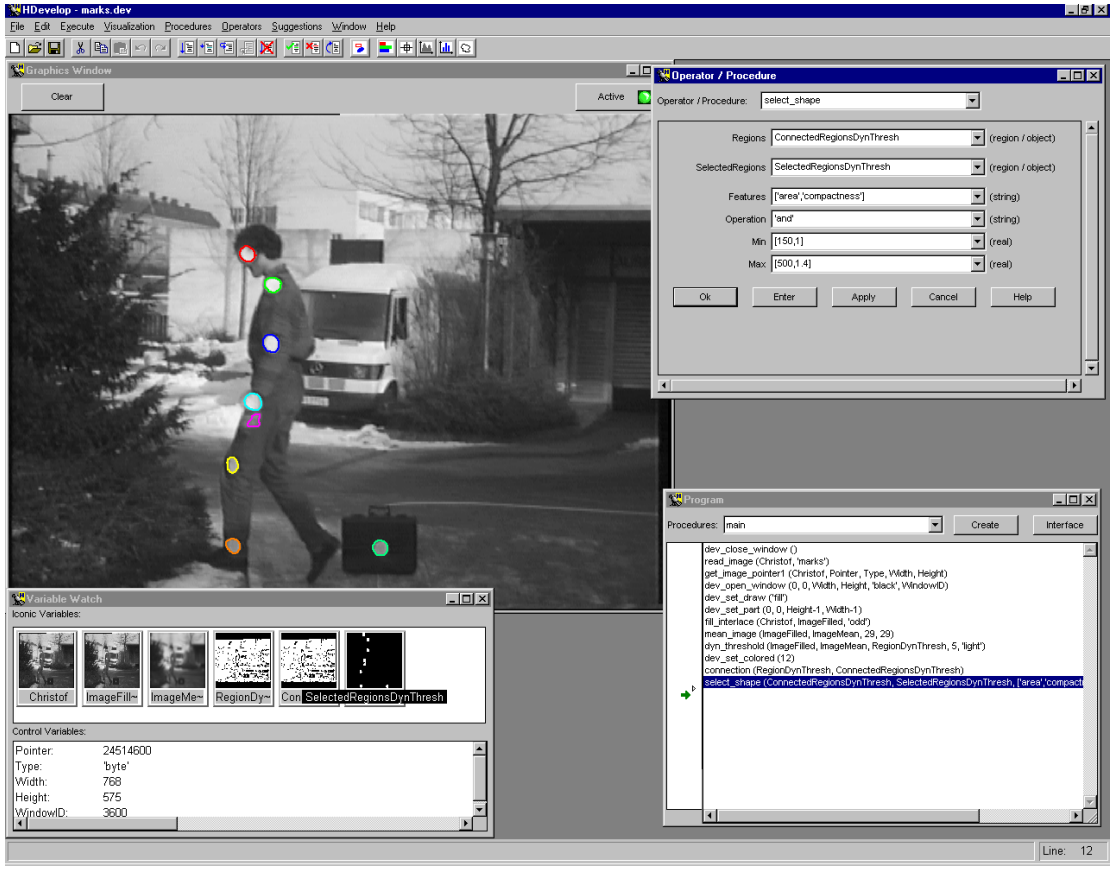

Figure 1.4: Region selection based on shape features.

displayed as a list (a tuple in HALCON terminology) in the variable window. Figure 1.5 shows the example session's result.

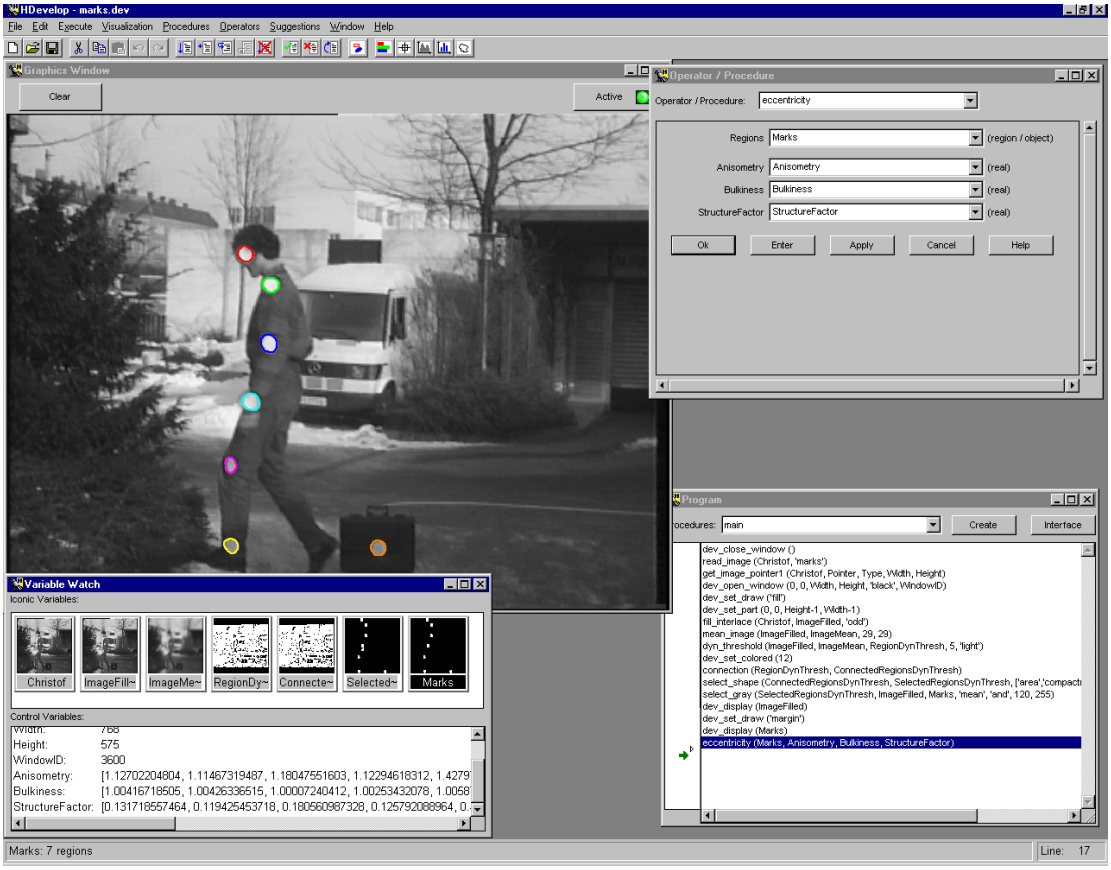

Figure 1.5: After calling the operator `eccentricity` the output parameters are displayed in the variable window in tuple notation.

Chapter 2

Graphical User Interface

HDevelop is an *integrated development environment (IDE)* to create machine vision programs. It comprises an *editor*, an *interpreter* with debug functions, a *management unit* for variables (iconic and control data) and extensive possibilities to visualize iconic data. You may use HDevelop for *rapid prototyping* as well as for developing complete programs. You create a program by choosing operators and specifying their parameters. To do so, you may use default values or values proposed by HDevelop. After having selected the appropriate parameters, you execute the operator and insert it into the program text (i.e., the *program window*). You can modify and verify your generated program interactively. All intermediate results (variables) are displayed graphically (images, regions and polygons (XLD)) or textually (numbers and strings).

After starting the HDevelop as described in section 1.3 on page 2, the main window will appear on your screen (see also figure 2.1 on page 14). It includes the following windows:

- a program window
- an operator window
- a variable window, and
- a graphics window.

In the following you will learn the functionality of these windows and their use while creating HDevelop programs.

Please note that in the UNIX environment the main window, the program window, and the operator window are combined into one window. Thus, there are only *three* windows.

2.1 Interacting with HDevelop

You interact with HDevelop through its graphical user interface. With the mouse you can manipulate visual controls such as menus or buttons in the HDevelop windows.

You can use the mouse as follows:

- Clicking the left mouse button once,
  you are able to *select* window-specific components, such as menu items, iconic variables, control variables, action buttons, checkboxes, and you give the insertion focus to a specific text field. Some of these text fields comprise a combo box which you may open in the same way.

  Furthermore, you select (invert) text in certain windows, e.g., in the program window. With this you are able to perform the general editor functions like `cut`, `copy` and `paste` (see section 2.3.4 on page 23 and section 2.3.12 on page 61).

  In the program window there is an extended mode to select lines by pressing the `<Shift>` or the `<Ctrl>` key during the mouse click. More than one line can be activated using the `<Shift>` key: All lines between the last activation and the new one will become activated. The `<Ctrl>` key is used to active or deactivate more than one line using single mouse clicks.

  Clicking at an item for the second time (after a short pause) will deactivate it (e.g., lines in the program window or variables in the variable window). Similarly, the activation passes to another item by clicking at it.

  Very important is the possibility to set the *program counter* (PC) at the left side of the program window (see section 2.4 on page 62 for detailed information). By combining a mouse click with special keys you can activate further functions:

  - Clicking the left mouse button once while pressing the `<Shift>` key:
    This places the insert cursor in the program window.
  - Clicking the left mouse button once while pressing the `<Ctrl>` key:
    A break point will bet set in the program window. By performing this action once more, the break point will disappear.

- Clicking the left mouse button twice
  results in an action that will be performed with the activated item. In the program window the operator or procedure call corresponding to the program line together with its parameters is displayed directly in the operator window and can then be modified.

  Iconic and control variables are displayed in the graphics window or in specific dialogs.

2.2 Procedures in HDevelop

As already mentioned in section 1.2 on page 2, every HDevelop program consists of one or more procedures; it always contains a main procedure. Once a procedure is created, calls to the procedure can be added to any program body just like operator calls. HDevelop has a runtime engine that makes it possible to execute procedure calls. HDevelop's runtime engine executes procedure calls as follows:

- When a procedure is called, an instance of that procedure or procedure call is created and pushed on the HDevelop internal call stack. If the called procedure has input parameters, its values are copied from the calling procedure, initializing the corresponding variables in the called procedure.
- Program execution is then continued at the first executable program line in the called procedure body and proceeds until the first return operator in the called procedure body is reached.

- By executing the return line, program execution returns to the calling procedure. If the called procedure has output parameters, they are copied to the corresponding variables of the calling procedure.

The mechanism of calling procedures in HDevelop works recursively, i.e. a procedure can call any other procedure, including itself. The main procedure has a special status inside the HDevelop runtime engine: it is called only once and automatically at the start of HDevelop and cannot be called by other procedures (including itself). This implies that the main procedure is always the top-most procedure in the procedure calling hierarchy and that it cannot be returned from.

The static part of a HDevelop program is defined by its procedures, with a procedure itself being described by its interface and program body plus its lists of local procedure body variables and procedure parameters. Additionally, the current status of the HDevelop runtime engine is defined by the call stack, which contains all procedure calls in reversed order, with a call to the main procedure always being the first procedure call on the stack. Program execution is generally continued in the top-most procedure call on the stack, which belongs to the last called procedure. Every procedure call is defined by the program counter or PC and by the values of the variables of the called procedure.

The HDevelop GUI is designed to display one procedure and procedure call at a time. The rest of this manual will refer to the currently displayed procedure and procedure call as to the current procedure and procedure call, respectively. The procedure or program body of the current procedure is always displayed in HDevelop's program window, while the interface of the current procedure can be viewed optionally in the procedure interface dialog (see section 2.4.3.1 on page 65). The current procedure variables are displayed in the variable window with procedure interface parameters being specially marked.

There are multiple ways how the current procedure and procedure call can change, all of which are going to be explained in more detail in the following sections. Here, it should only be mentioned that the current procedure can be changed directly through the corresponding GUI control elements as well as programmatically when program execution is stopped in a procedure, which then automatically becomes the current procedure. Depending on whether the current procedure has no, one, or multiple calls on the stack, certain features of the HDevelop GUI may be enabled or disabled. In case the current procedure is not called, i.e., has no procedure call on the stack, execution inside the procedure is not allowed and its variables are always uninitialized. If the current procedure has multiple calls on the stack, it cannot be edited in order to avoid inconsistent states of the call stack. Keep in mind that a procedure call is defined by its PC and variable instances, which most likely will differ if the current procedure has multiple calls on the stack. Relating to the HDevelop GUI, the static parts of the current procedure, which are the program body, insert cursor and variable lists, are displayed independently of the state of the procedure, whereas the displayed PC and variable values depend on the current procedure call.

2.3 Main Window

On a Windows NT/2000/XP system, the *main window* contains the other four HDevelop windows and possibly additional graphics windows and dialogs. In contrast, on a UNIX system, the main window comprises the program window and the operator window.

The main window can handle HDevelop programs, manipulate the graphics output, offer all HALCON and HDevelop operators and procedures, give suggestions and help on choosing operators and manage

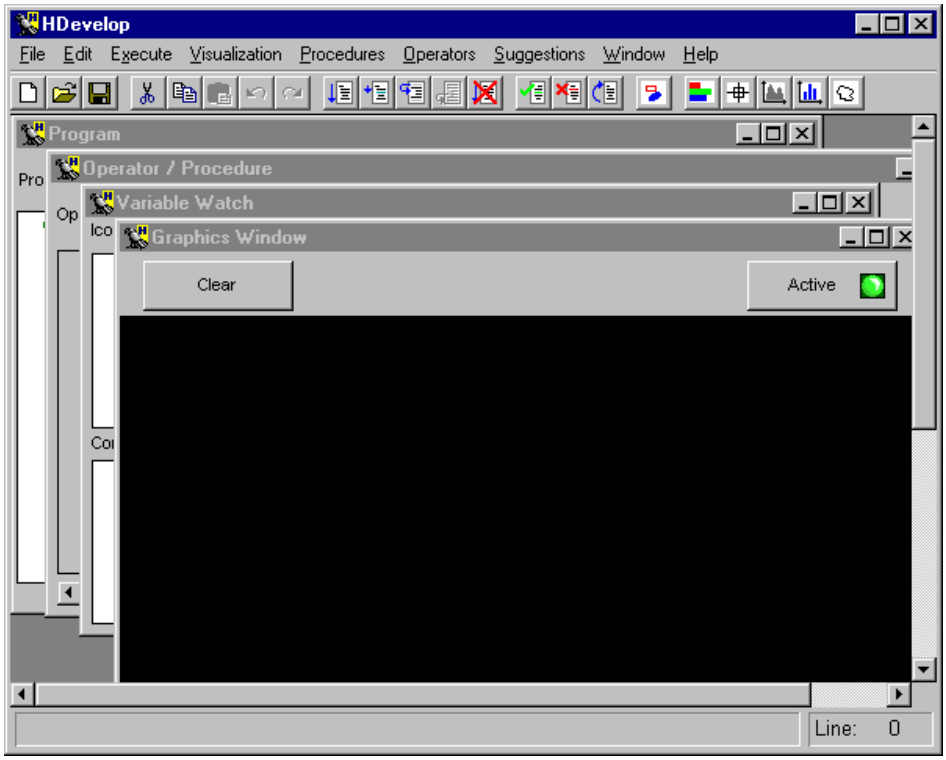

Figure 2.1: The main window (Windows NT/2000/XP).

the HDevelop windows. After starting HDevelop (on a Windows NT/2000/XP system), you will see a window configuration similar to figure 2.1.

The main window comprises five areas:

- a title bar,
- a menu bar,
- a tool bar,
- a window area, and
- a status bar.

In the following sections you will find all necessary information to interact with this window.

2.3.1 Title Bar

Your HDevelop main window is identified by the title `HDevelop` in the window's title bar. After loading or saving a file, the file name will be displayed in the title bar. Additionally, it offers three buttons on the right hand side to iconify and to maximize the window, and to exit the HDevelop session.

2.3.2 Menu Bar

In the menu bar of the main window HDevelop functionality is offered. Here, you may perform the important actions to solve your image processing tasks, i.e., choose HALCON or HDevelop operators or procedures or manipulate the graphical output. Every menu item opens a *pull-down* menu (henceforth abbreviated as menu) with optional submenus. You open a menu by clicking a menu item (inside the appropriate text) or via the keyboard (by pressing the key `<Alt>` in combination with the underlined letter of the menu item). All menu items are going to be explained in the following.

2.3.3 Menu `File`

In the menu `File` you will find all functions to load an image and existing programs and to save recently created or modified programs, respectively. Furthermore, you may export HDevelop programs to C++, C, or Visual Basic, and also print them. Figure 2.2 shows all the functions in this menu.

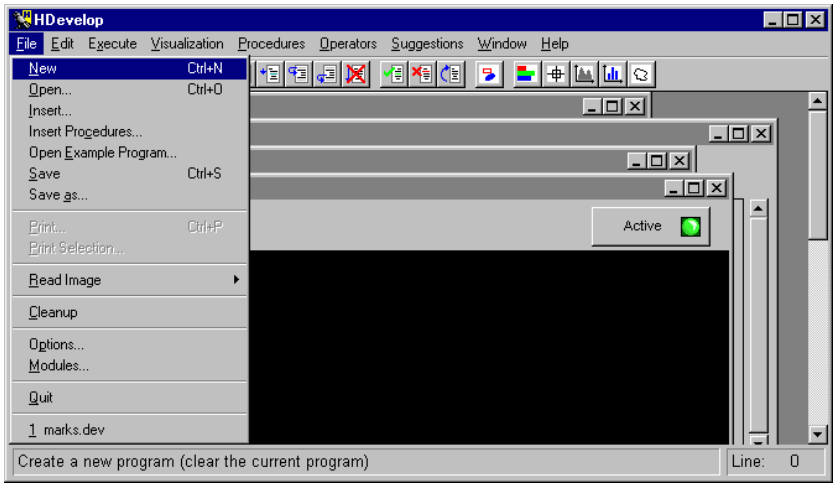

Figure 2.2: The menu `File`.

2.3.3.1 `File` ▷ `New`

The menu item `File` ▷ `New` (keyboard shortcut `<Ctrl> N`) deletes the current program including all procedures. The contents of variables are deleted before removing them. In addition, all graphics windows except one are closed. The last window will be cleared. The display parameters for the remaining graphics window are identical to those when starting HDevelop. The first four parameters of the menu `File` ▷ `Options` are reset to their initial state: The update of windows, variables, PC, and time is on.

A security check prevents you from deleting the current program accidentally if the program has not been saved. A dialog box appears and asks whether you want to save the HDevelop program before its dismissal. If you choose to do so and a file name is already specified, the current program is saved to that file, otherwise a file selection dialog is invoked where you can determine a corresponding file. You can also choose to dismiss the current program without saving the changes, or cancel the action.

2.3.3.2 `File` ▷ `Open`, `File` ▷ `Insert`

By clicking on the menu `File` ▷ `Open` (keyboard shortcut `<Ctrl> O`), you can load an existing HDevelop program. Alternatively, you can select `File` ▷ `Insert` to insert a file into the current program body at the line in which the insert cursor is located. In both cases, a dialog window pops up and waits for your input (see figure 2.3). It is called `Open HDevelop Program File`. Please note that text, Visual Basic, C, or C++ versions of a file cannot be loaded.

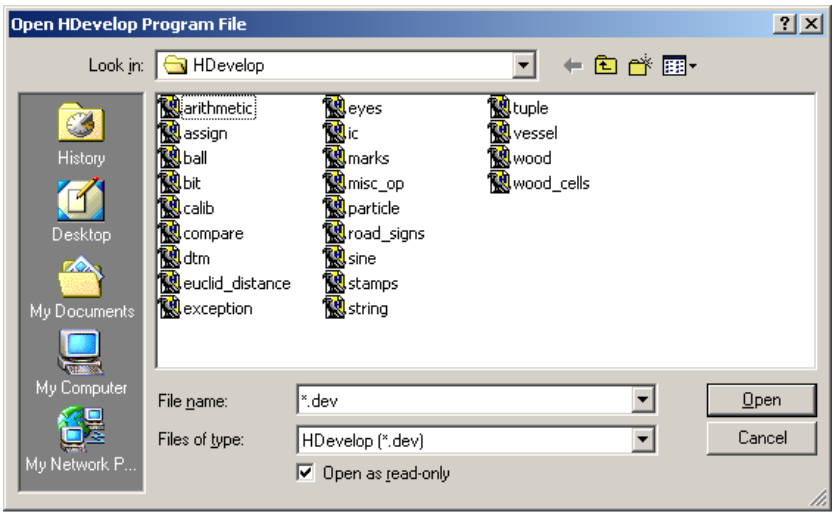

Figure 2.3: The dialog window to open an HDevelop file.

In the top-most text field you may specify a directory which contains your HDevelop programs. A combo box at the right hand side helps you browsing your directories. To move one directory level up, you press the button on the right hand side of this text field. The next button creates a new folder to store HDevelop programs. By pressing the last button you can activate or deactivate the option to see more details about your HDevelop programs, i.e., the program size, the program type, the date when the most recent user update occurred, and file attributes.

The middle text area displays all available HDevelop files to choose from. By clicking the left mouse button on a file name you select it. Double-clicking a file name opens the file immediately and displays it in the program window (see section 2.4 on page 62).

Furthermore, you may specify the file name in the text field below. The combo box for file type has no effect because only HDevelop programs with the extension `.dev` can be loaded. If you want to open your file with a write protection, choose the checkbox at the bottom of this dialog window. To open your specified file, you press the button `Open`. This action deletes an already loaded HDevelop program and all created variables. The same actions as with `File` ▷ `New` are performed. Now you can see the main procedure body of your new program in the program window. The file name is displayed in the title bar of the main window. All its (uninstantiated) variables are shown in the variable window. To indicate that they do not have any computed values, the system provides the iconic and control variables with a question mark. The program counter is placed on top of the program body and you are ready to execute the program. The visualization and options will be reset after loading (same as `File` ▷ `New`).

You can cancel this task by pressing the corresponding button. By using one of the two buttons `Open` or `Cancel` the dialog window disappears.

After you have loaded a program, the corresponding file name will be appended at the bottom of the menu `File`, in the file history. This allows you to switch between recently loaded files quickly. The most recently loaded file is always listed first.

2.3.3.3 `File` ▷ `Insert Procedures`

Via this menu item you can add procedures from a HDevelop program file to the current program. All procedures except the main procedure are loaded from the selected file (see figure 2.3). If the current program already contains a procedure with the same name, the newly added procedure will be renamed by appending a suffix to its name.

2.3.3.4 `File` ▷ `Open Example Program`

Selecting this menu item opens a dialog that allows you to load HDevelop example programs grouped by different topics and categories (see figure 2.4). The dialog is made up of three separate list boxes. The left list contains the three available topics 'Industry', 'Application Area' and 'Method'. If you select a topic all categories of that topic are listed in the middle part of the window. If you select one of the categories all corresponding example programs belonging to the topic and category are displayed in the right subwindow. Single clicking on an example program in the right list displays a short information in HDevelop's status bar, while double clicking or clicking `OK` will load the selected program and close the dialog. A HDevelop example program can appear under different topics and categories.

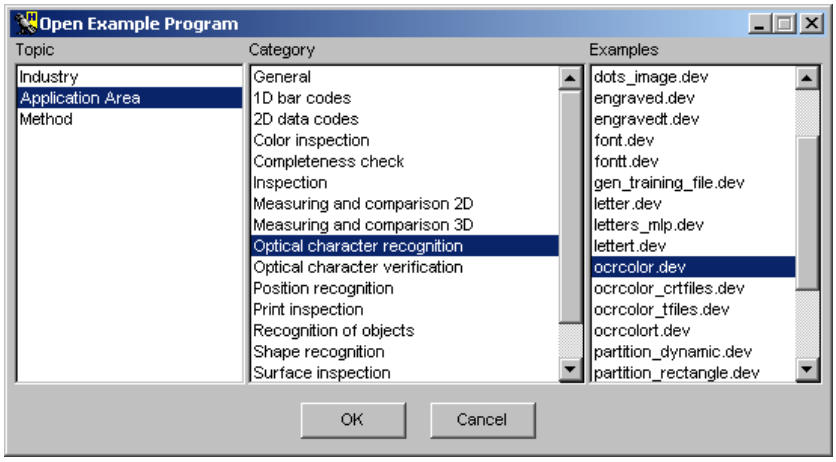

Figure 2.4: HDevelop example programs grouped by topics and categories.

2.3.3.5 File ▷ Save

The menu item File ▷ Save (keyboard shortcut <Ctrl> S) saves the current program to a file. If no file name has been specified so far, the dialog corresponding to File ▷ Save As will be activated.

2.3.3.6 File ▷ Save As

The menu item File ▷ Save As saves the current program to a file. The type of file (HDevelop, text, Visual Basic, C, or C++) can be selected (see figure 2.5).

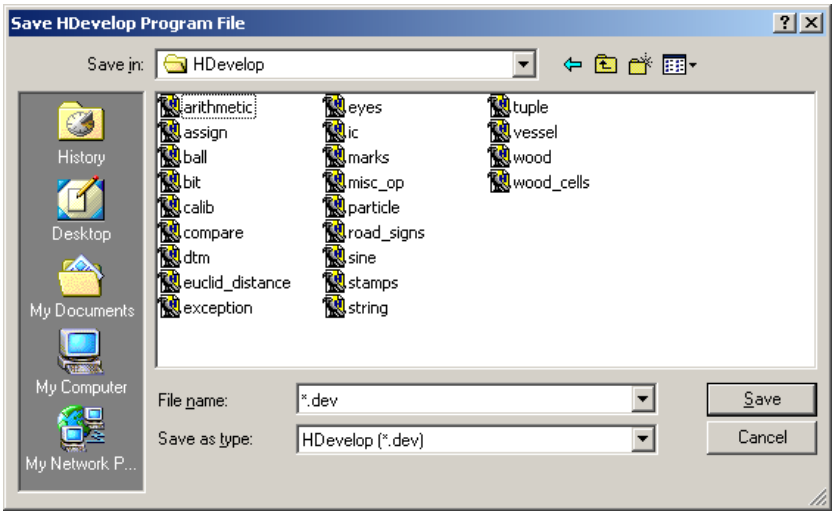

Figure 2.5: The dialog window to save a program to a file.

A dialog box (similar to the window described in menu item File ▷ Open) is opened, in which you can specify the appropriate settings. You may specify a new file name and a directory where to locate this file. By clicking the combo box of the text field called Files of type, you may indicate whether the HDevelop program remains a HDevelop program or is exported as a C++, C, or Visual Basic program, or is transformed into an ASCII file. In UNIX you select the file type by entering the corresponding file extension manually. For C++ code you have to add .cpp to the file name, for C code .c, and for ASCII .txt. The extension for Visual Basic is .bas. Default type is the HDevelop type (extension .dev). The details of code generation are described in chapter 4 on page 105.

Similar to loading, the file name of the program you save is appended at the end of the menu File.

2.3.3.7 File ▷ Print

The menu item File ▷ Print (keyboard shortcut <Ctrl> P) enables you to print the current program. Upon selecting the menu item, a dialog appears in which you can configure the printing process.

2.3.3.8 File ▷ Print Selection

In contrast to the menu item File ▷ Print, the menu item File ▷ Print Selection prints only the selected part of the current procedure body.

2.3.3.9 File ▷ Read Image

The menu item File ▷ Read Image contains several directories from which images are usually loaded. The first entry of this menu always is the directory from which the most recent image was loaded. This is useful when several images from a non-standard directory must be read. The remaining entries except the last one are the directories contained in the environment variable %HALCONIMAGES%. The final directory, denoted by '.', is the current working directory of the HDevelop program, which usually will be %HALCONROOT% on Windows systems, and the directory in which HDevelop was started on UNIX systems.

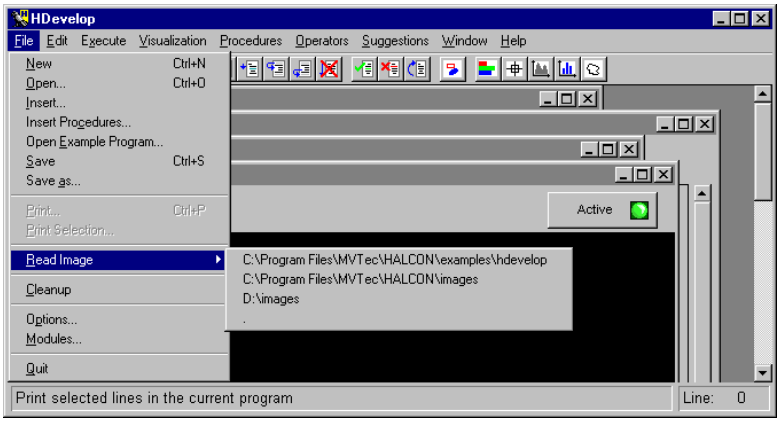

Figure 2.6: The menu item File ▷ Read Image.

When any of the directories is selected, an image file selection box appears. Again, its functionality is similar to the dialog described in menu item File ▷ Open. Figure 2.7 shows an example of this dialog.

After selecting a file name, the name of the variable for the image in the current HDevelop procedure has to be selected. To do this, a small dialog appears after pressing Open or double clicking a file. For easy handling, HDevelop suggests a name derived from the selected file name. You may adopt or edit this name. If you want to use a name of an already created iconic variable, a combo box offers you all iconic variable names. To do so, you click the button on the right side of the text field. Note that the reuse of a variable name deletes the old content and replaces it with the new image.

2.3.3.10 File ▷ Cleanup

The menu item File ▷ Cleanup deletes all unused variables (iconic and control data) from the current procedure. These are variables in the variable window that are no longer used in any operator or procedure call in the current procedure body. This can happen after the deletion of program lines or after

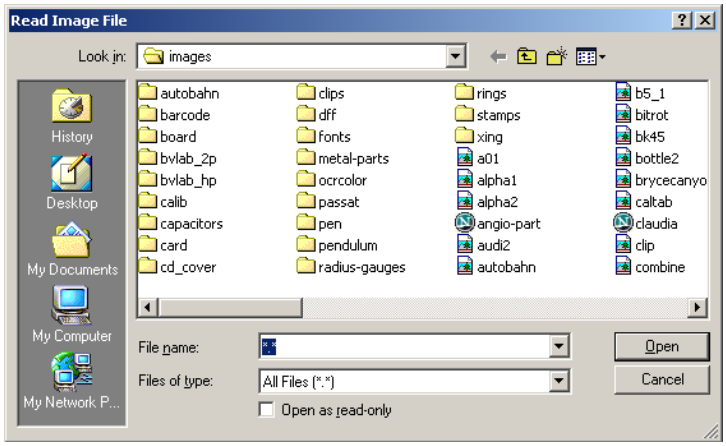

Figure 2.7: The dialog window to load an image.

editing variable names, because the corresponding variables are not deleted automatically. You may use this menu item during a longer editing process to reorganize your variable window (see also section 2.6 on page 75).

2.3.3.11 `File` ▷ `Options`

The menu item `File` ▷ `Options` opens a control window, which you can use to modify output behavior during runtime (see figure 2.8).

- `Update PC`
  The first item (see page 62) concerns the display of the current position while running the program. The so called *PC* (program counter) always indicates the line of the currently executing operator or procedure call or the line before the next operator or procedure call to execute. Using the PC in this way is time consuming. Therefore, you may suppress this option after your test phase or while running a program with a lot of "small" operators inside a loop.

- `Update Variables`
  This checkbox concerns the execution of a program: Every variable (iconic and control) is updated by default in the variable window (see page 75). This is very useful in the test phase, primarily to examine the values of control data, since iconic data is also displayed in the graphics window. If you want to save time while executing a program with many operator calls you may suppress this output. Independent of the selected mode, the display of all variables will be updated after the program has stopped.

- `Sort Variables`
  This checkbox determines whether the iconic and control variables displayed in the variable window will be sorted by name.

- `Update Window`
  This item concerns the output of iconic data in the graphics window after the execution of a

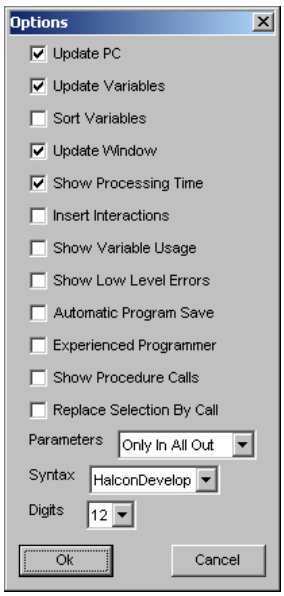

Figure 2.8: The options window.

HALCON operator. With the default settings, all iconic data computed in the run mode (see section 2.3.5.1 on page 26) is displayed in the current graphics window. You may want to suppress this automatic output, e.g., because it slows down the performance. If the output is suppressed you have the same behavior as exported C, C++, or Visual Basic code, where automatic output of data is not supported.

- `Show Processing Time`
  This checkbox indicates whether the required runtime of the last operator or procedure call should be displayed after the execution has stopped. It is a measurement of the needed time for the current operator or procedure call (without output and other management tasks of HDevelop). Along with the required runtime, the name of the operator or procedure is displayed in the status bar at the bottom of the main window. Please note that the displayed runtime can vary considerably. This is caused by the inaccuracy of the operating system's time measurement procedure.

- `Insert Interactions`
  Sometimes it is very helpful to record user interactions as a sequence of operators in the program. To do so, you select this option. From now on interactions are inserted as a program line in the program window. For example, selecting the graphic color red by choosing the appropriate menu inserts the program line

  ```
  dev_set_color('red')
  ```

  into the program window.

- `Show Variable Usage`
  If you activate a variable (by single-clicking on it) all lines in the program that contain the variable are marked on the left with a black frame. This works with iconic and control variables. You can

activate one iconic and one control variable simultaneously. Each activated variable is marked by a black background for the name in the variable window.

- `Show Low Level Errors`
  Low-level errors of HALCON are normally invisible for the user because they are transferred into more comprehensive error messages or simply ignored. Activating this item generates a message box each time a low-level error occurs.

- `Automatic Program Save`
  If you activate this option, the program is automatically saved before each execution of the program, i.e., before a Run or Step operation. The file name the program is saved to is the file name of the current program. Therefore, if you create a new program you must first select `File` ▷ `Save As` manually to give the program a file name.

- `Experienced Programmer`
  If this option is activated, the internal temporary memory usage of the last operator or procedure call is displayed next to the required runtime. Also, the lists of the parameter combo boxes in the operator window are extended so that they include variables whose semantic types do not match the semantic types of the corresponding parameters of the selected operator. This option is more suitable for experienced HDevelop users.

The following elements concern HDevelop procedures. Detailed information can be found in section 2.4.3 on page 65:

- `Show Procedure Calls`
  If this option is selected procedure calls are marked in the program window with a green bar.

- `Replace Selection By Call`
  If this option is switched on selected program lines, from which a procedure is created, are automatically replaced by a call to the created procedure.

- `Parameters`
  Via this combo box you can choose between different modes for the automatic creation of a procedure interface.

The final elements customize the display of the program and of numeric variables:

- `Syntax`
  Using a combo box, you may specify the output mode inside the program window. Depending on the mode, each HALCON or HDevelop operator or procedure call is shown in a specific syntax like `HalconDevelop` (default syntax) or `C`.

- `Digits`
  With this checkbox, you can control how many digits of floating point numbers are displayed in the variable window. The selected number is the total number of digits displayed. Therefore, if you have selected four digits, the result of the following assignment

  ```
  assign (4*atan(1), PI)
  ```

  is displayed as 3.142. Note that the changes do not take effect until the values of the variables are actually updated by running the program, i.e., the variables are not redisplayed automatically.

Before continuing your HDevelop session, you have to close the option window by pressing the button `Ok` or by cancelling the action. If `Insert Interactions` is activated, the changes applied inside the dialog will result in automatic operator insertion *after* pressing `Ok`.

HDevelop saves the current selections for the options "Show Variable Usage", "Show Low Level Errors", "Automatic Program Save", "Experienced Programmer", "Syntax", and "Digits", and restores them upon start. Under Windows, the options are stored in the registry; under UNIX, the options are stored in the subdirectory `.hdevelop` of the directory referenced by the environment variable `$HOME`.

2.3.3.12 `File` ▷ `Modules`

The menu item `File` ▷ `Modules` opens a window, in which the HALCON modules used by the current program are displayed (see figure 2.9). This window allows you to get an estimate of how many modules your application would need in a runtime license. Please refer to the Installation Guide, section 3.4 on page 25, for more information about modules and runtime licenses.

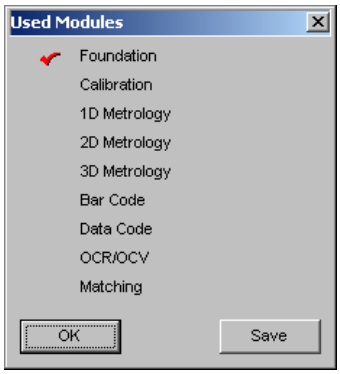

Figure 2.9: The modules window.

By clicking `Save`, the modules required by the current program are saved in a file with the extension `.mod` in the currently used directory.

2.3.3.13 `File` ▷ `Quit`

The menu item `File` ▷ `Quit` terminates HDevelop.

2.3.3.14 `File` ▷ `History`

At the bottom, the menu `File` displays the most recently loaded files.

2.3.4 Menu `Edit`

In this menu you find all necessary functions to modify the current HDevelop procedure body displayed in the *program window* (see section 2.4 on page 62).

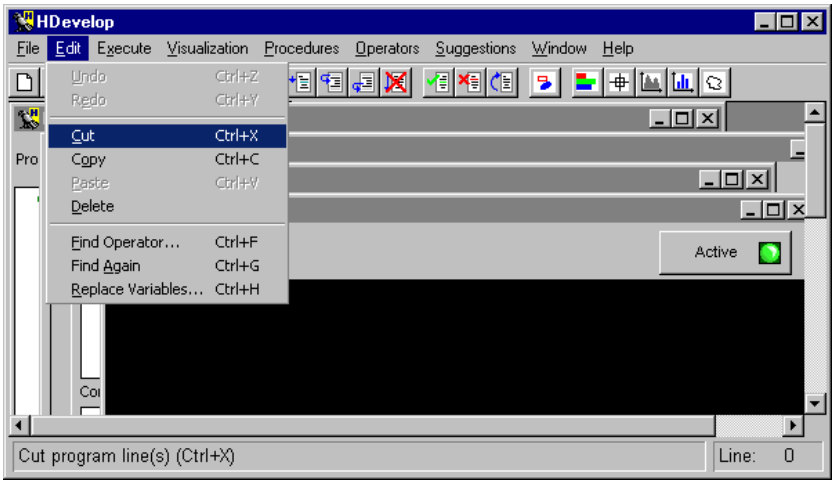

Figure 2.10: The menu item `Edit`.

2.3.4.1 `Edit` ▷ `Undo`

You may undo your previous activities via the menu item `Edit` ▷ `Undo`. For example, by selecting it three times you cancel the last three user actions. The procedure to which the last undo belongs becomes the current procedure.

2.3.4.2 `Edit` ▷ `Redo`

You can also revoke undo activities by selecting `Edit` ▷ `Redo`. This is a quick way to restore the state before the last undo operation.

2.3.4.3 `Edit` ▷ `Cut`

You may use the items `Edit` ▷ `Cut`, `Edit` ▷ `Copy`, and `Edit` ▷ `Paste` for changing the procedure body contents. First you have to select the part of the program (at least one program line) that has to be changed (use the left mouse button). To delete this part, select the item `Edit` ▷ `Cut` (keyboard shortcut `<Ctrl> X`).

The deleted program part is stored in an internal buffer. By using the item `Edit` ▷ `Paste` (keyboard shortcut `<Ctrl> V`), the buffer remains unchanged.

2.3.4.4 `Edit` ▷ `Copy`

By selecting `Edit` ▷ `Copy` (keyboard shortcut `<Ctrl> C`), you store the selected program lines directly in an internal buffer. Additionally, for every procedure call line the corresponding procedure and all

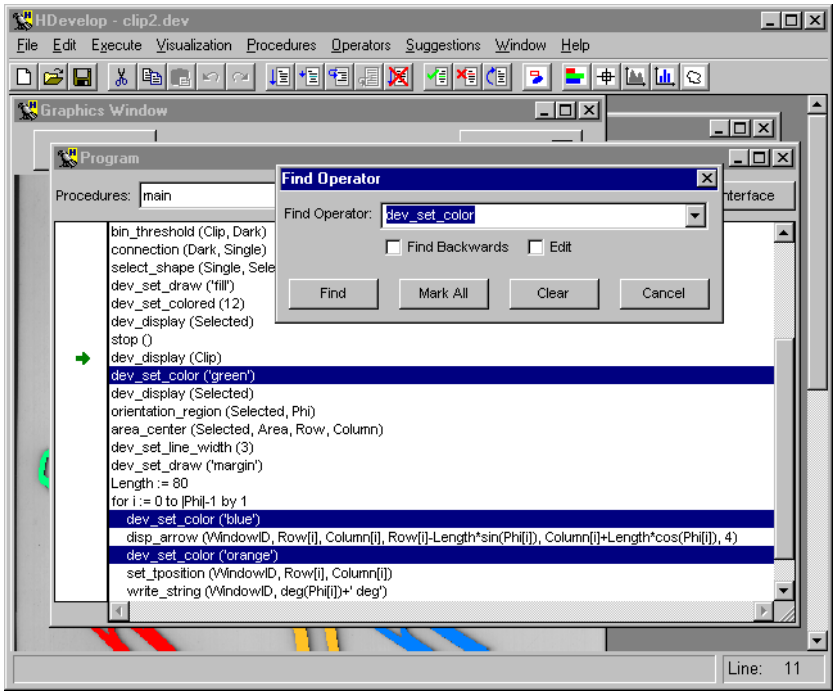

Figure 2.11: Searching for an operator.

procedures that can be reached from it are copied to the buffer. This is necessary in order to obtain a consistent program when pasting procedure call lines to a program in which the corresponding procedures might not exist.

2.3.4.5 `Edit` ▷ `Paste`

To insert this buffer in the current HDevelop procedure body you place the insertion cursor at the desired position and then select the menu item `Edit` ▷ `Paste` (keyboard shortcut `<Ctrl> V`). If the buffer contains procedures that do not exist they are added to the current program. The mechanism of copying and pasting procedure call lines together with the corresponding procedures is an easy way to transfer procedures between different HDevelop programs.

2.3.4.6 `Edit` ▷ `Delete`

The menu item `Edit` ▷ `Delete` deletes all selected program lines without storing them in an internal buffer. The only way to get the deleted lines back in your procedure body is to use the item `Edit` ▷ `Undo`.

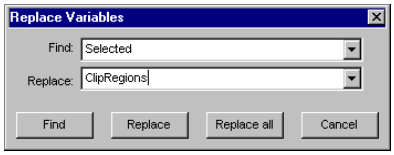

Figure 2.12: Replacing variables.

2.3.4.7 `Edit` ▷ `Find Operator`

The menu item `Edit` ▷ `Find Operator` (keyboard shortcut `<Ctrl> F`) allows to search for the appearance of an operator or procedure call in the current procedure body. Figure 2.11 depicts the corresponding dialog: Specify an operator or procedure name in the text field, then find the next appearance by clicking the button `Find`, or the previous one by checking the box `Find Backwards`. The figure shows the result of clicking `Mark All`. If you check the box `Edit`, the operator window pops up and lets you edit the found operator or procedure calls.

2.3.4.8 `Edit` ▷ `Find Again`

The menu item `Edit` ▷ `Find Again` (keyboard shortcut `<Ctrl> G`) repeats the search specified via the menu item `Edit` ▷ `Find Operator`.

2.3.4.9 `Edit` ▷ `Replace Variables`

The menu item `Edit` ▷ `Replace Variables` (keyboard shortcut `<Ctrl> H`) allows to search and replace variable names in the current procedure body. Figure 2.12 depicts the corresponding dialog: After specifying the variable name to find and the name to replace it by, you can let HDevelop replace all instances of the variable by clicking `Replace All`. If you click `Find`, the next instance of variable is displayed in the program window; you can then replace its name by clicking `Replace`.

2.3.5 Menu `Execute`

In this menu item you find all necessary functions to execute a HDevelop program. As mentioned in section 2.2 on page 12, program execution is always continued at the top-most procedure call, which in most cases corresponds to the current procedure call. The procedure body displayed in the *program window* (see section 2.4 on page 62) belongs to the current procedure.

2.3.5.1 `Execute` ▷ `Run`

If you select `Execute` ▷ `Run` (keyboard shortcut `F5`), HDevelop executes your program starting at the *PC*'s position in the program window. All following program lines are going to be performed until the end of the current program. After the execution is finished, the main procedure becomes the current

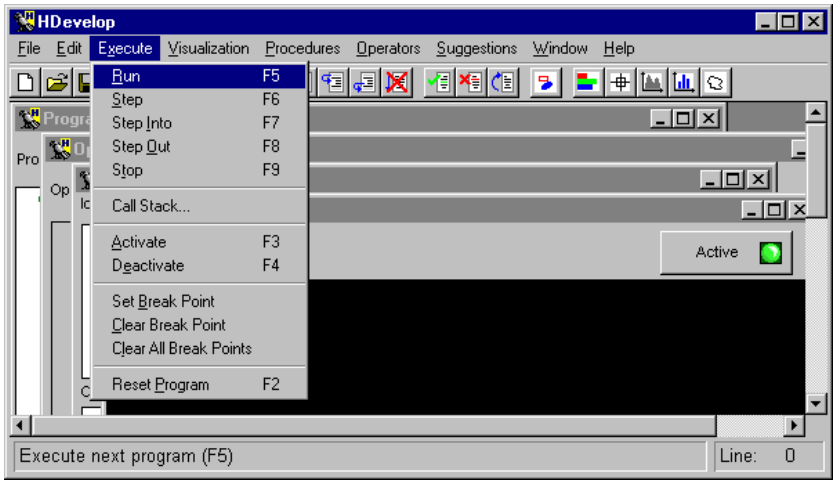

Figure 2.13: The menu item `Execute`.

procedure. Note that a break point, stop instruction, or runtime error may interrupt the execution of your program.

During the execution of operator or procedure calls the following special behavior occurs:

- Although the mouse pointer indicates that HDevelop is not ready to handle user input (clock-shaped mouse pointer), you may initiate limited activities. For example, if you double-click variables in the *variable window* (see section 2.6 on page 75) they will be visualized; you may modify parameters for the graphics windows as described in section 2.3.6 on page 30; you may even modify the current procedure body. Note that all user interaction except `Execute` ▷ `Stop` is disabled during program execution in case the latter was not started in the main procedure. HDevelop may be slow to react to your actions while the program is running. This is caused by the fact that HALCON reacts to user input only *between* calls to operators.
- A variable window update during runtime will only be performed if it has not been suppressed (see section 2.3.3.11 on page 20). In any case, the values of all variables are shown in the variable window after the execution's termination.

While the program is running, the menu items `Execute` ▷ `Run`, `Execute` ▷ `Step`, `Execute` ▷ `Step Into`, and `Execute` ▷ `Step Out` (and the corresponding toolbar buttons) are grayed out, i.e., you cannot execute them.

You have the following possibilities to stop your HDevelop program:

1. The program runs until the last operator or procedure call in the current program (i.e. the main procedure body) has been called. The *PC* is positioned behind this operator. This is the usual way to terminate a program.
2. The menu `Execute` ▷ `Stop` (or the corresponding toolbar button) has been pressed.
3. A break point has been set (see section 2.4 on page 62). In this case, the last operator or procedure call that will be executed is the one *before* the break point.

4. Menu item File ▷ Quit has been executed (see section 2.3.3.13 on page 23).
5. A runtime error occurred. An input variable without a value or values outside a valid range might be typical reasons. In this case the PC remains in the line of the erroneous operator or procedure call.
6. A stop instruction is executed. The PC remains on the next executable line after the stop instruction.

Note that the procedure and procedure call in which program execution was stopped automatically become the current procedure and procedure call.

2.3.5.2 Execute ▷ Step

Selecting Execute ▷ Step (keyboard shortcut F6) enables you to execute a program (even if it is not complete) step by step. HDevelop executes the operator or procedure call directly to the right of the *program counter* (PC, indicated by a green arrow, see section 2.4 on page 62). The mouse pointer changes its shape to a clock. This indicates that HDevelop is active and not available for any user input. After the operator or procedure call has terminated, all computed values are assigned to their respective variables that are named in the output parameter positions. Their graphical or textual representation in the variable window is also updated. If iconic data has been computed, you will see its presentation in the current graphics window. In the status bar of the program window the runtime of the operator or procedure call is indicated (if the time measurement has not been deactivated).

The PC is then moved to the next operator or procedure call to execute. If the operators or procedure calls are specified in a sequential order, this is the textual successor. In case of control statements (e.g., `if ... endif` or `for ... endfor`), the PC is set *on* the end marker (e.g., `endif` or `endfor`) after the execution of the last operator or procedure call inside the statement's body. After `endfor` and `endwhile`, the PC is always set on the beginning of the loop. If a condition (as `if` or `while`) evaluates to `FALSE`, the PC is set *behind* the end marker.

Suggestions in the menu Suggestions are determined for the recently executed operator. Finally, the mouse pointer's shape switches to the arrow shape and HDevelop is available for further transactions. Any user input which has been made during execution is handled now.

2.3.5.3 Execute ▷ Step Into

Execute ▷ Step Into (keyboard shortcut F7) allows you to step into procedure calls. Executing Execute ▷ Step Into with the PC on a procedure call line causes the corresponding procedure and procedure call to become the current procedure and procedure call, respectively. The PC is set on the first executable program line in the new current procedure. Execute ▷ Step Into has the same effect as Execute ▷ Step if the program line to be executed is not a procedure call.

2.3.5.4 Execute ▷ Step Out

Selecting Execute ▷ Step Out (keyboard shortcut F8) steps out of the current procedure call. Program execution is continued until the first executable program line after the previous procedure call in the calling procedure is reached. The previous calling procedure becomes the current procedure. Execute ▷ Step Out has no effect when called in the main procedure.

2.3.5.5 Execute ▷ Stop

You may terminate the execution of a program by selecting Execute ▷ Stop (keyboard shortcut F9). If you do so, HDevelop continues processing until the current operator has completed its computations. This may take a long time if the operator is taking a lot of time to execute. There is no way of interrupting a HALCON operator. The procedure and procedure call in which program execution was stopped becomes the current procedure and procedure call, respectively. After interrupting a program you may continue it by selecting Execute ▷ Run or Execute ▷ Step. You may also edit the program before restarting it (e.g., by parameter modification, by exchanging operators with alternatives, or by inserting additional operators).

2.3.5.6 Execute ▷ Call Stack

Selecting this item depicts a dialog (see figure 2.14) that contains a list of the names of all procedures that are currently called on HDevelop's internal call stack. The top-most procedure call belongs to the most recently called procedure, the bottom-most procedure call always belongs to the main procedure. Double clicking on a procedure call in the dialog makes the selected procedure call the current procedure call and thus the procedure belonging to the selected procedure call the current procedure.

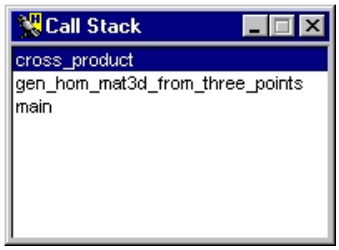

Figure 2.14: Callstack dialog.

2.3.5.7 Execute ▷ Deactivate, Execute ▷ Activate

During testing, it is often useful to prevent some lines of the program from being executed. This can be achieved by selecting the appropriate lines in the program window and then selecting Execute ▷ Deactivate. Then, an asterisk is placed on the beginning of the selected lines, i.e., the lines appear as comments in the program window and have no influence on the program during runtime.

The deactivated lines are still part of the program, i.e., they are stored like all other lines and their variables are still needed like all other variables. To reverse this action, select Execute ▷ Activate.

Note that you can insert a 'real' comment into your program by using the operator comment.

2.3.5.8 Execute ▷ Set Break Point, Execute ▷ Clear Break Point

These menu items set or clear a break point on the line(s) that are currently selected in the program. In most cases, however, it is easier to set and clear individual break points pressing the left mouse button

and the <Ctrl> key in the left column of the program window as described in section 2.4 on page 62.

2.3.5.9 Execute ▷ Clear All Break Points

With this menu item you can clear all break points in the current HDevelop program.

2.3.5.10 Execute ▷ Reset Program

With the menu item Execute ▷ Reset Program, you can reset the current HDevelop program to its initial state. The main procedure becomes the current procedure and the call stack is cleared of all procedure calls except the main procedure call. The latter is reset, i.e. all variables have undefined values and the *program counter* is set to the first executable line of the main procedure. The *break points*, however, are not cleared. This menu item is useful for testing and debugging programs.

2.3.6 Menu Visualization

Via this menu, you can open or close graphics windows and clear their displays. Furthermore, you may specify their output behavior during runtime. All items which can be selected are shown in figure 2.15.

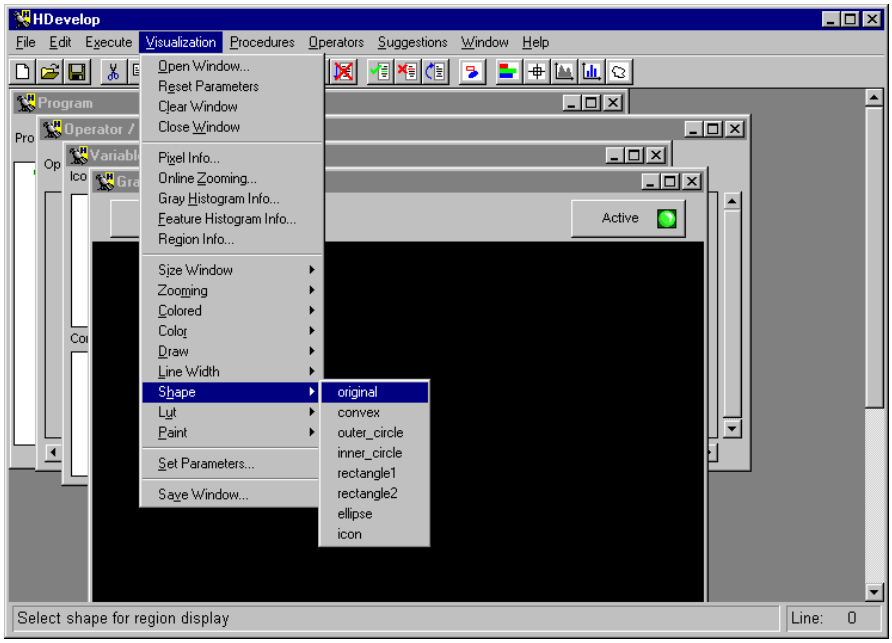

Figure 2.15: The menu Visualization.

2.3.6.1 Visualization ▷ Open Window

By using this menu item, you open an additional[1] graphics window.

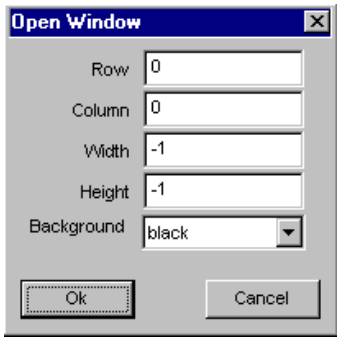

Figure 2.16: Dialog window of menu item Visualization ▷ Open Window.

For this, a dialog window pops up (see figure 2.16). Here, you may specify some graphics windows attributes. The position, size and background color of the new graphics window can be specified. For example, it is more convenient to have a white background while building graphics for slides or reports (see the HALCON operator dump_window). If the window height and width are set to -1, the window has the same size as the largest image in the current session. A position value of -1 specifies that the window position is determined by the window manager (UNIX). If you have not already created an image, the size 512×512 is used.

The handling of graphics windows is described in more detail in section 2.7 on page 78.

2.3.6.2 Visualization ▷ Reset Parameters

Here, the display parameters of all graphics windows are set to their initial state (the state after starting the program). The only exception is the history of previously displayed objects and the size of each window. To clear the history you can use Visualization ▷ Clear Window, to set the size Visualization ▷ Size Window.

2.3.6.3 Visualization ▷ Close Window

Selecting this item closes the active graphics window.

2.3.6.4 Visualization ▷ Clear Window

Via this menu item the active graphics window is cleared. The history (previously displayed objects) of the window is also removed.

[1] Normally upon starting, HDevelop automatically opens one graphics window.

2.3.6.5 `Visualization` ▷ `Pixel Info`

Here, you can open an inspection display as depicted in figure 2.17. This is used for interactive examination of gray values of images. Apart from this, the size, pixel type, and the number of channels are displayed.

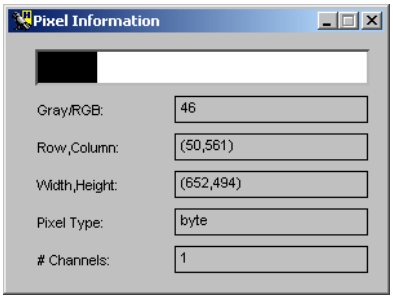

Figure 2.17: Online gray value inspection and basic image features.

The upper part of the dialog contains a gauge to display the gray value graphically. The range goes from 0 (left) to 255 (right). Normally, the gray value of the first channel is displayed with a black bar. For color images in RGB-space (three channels with red, green, and blue values) three colored bars are used. If the gray value is below 1, the gauge is white (background). If the value is above 255, the gauge is black or colored for RGB images.

Below the gauge, the gray values are displayed as numbers. If more than three channels are present, only the gray value of the first channel is displayed.

Below the gray values, the coordinates of the mouse position is displayed. Below these, the size, pixel type, and the number of channels are shown.

2.3.6.6 `Visualization` ▷ `Zooming`

With this menu item, a tool for real-time viewing of zoomed parts of an image object is opened. Figure 2.18 shows the layout of the real-time zooming window.

The upper part of the tool contains a window of fixed size 256×256, in which the part of the graphics window, over which the mouse pointer is located, is displayed enlarged. In the zooming window, this pixel is marked by a red square; its coordinates are displayed at the bottom of the zooming window.

The factor, by which the enlargement is done can be adjusted with the combo box `Zooming factor`. A zooming factor of 0 corresponds to displaying the contents of the graphics window in to normal resolution, i.e., one pixel in the image object corresponds to one pixel in the zooming window. Increasing the zooming factor by 1 roughly increases the enlargement by a factor of 2.[2]

[2] Yes, only roughly by a factor of 2, since the image is scaled such that the red square that indicates the mouse pointer position is located in the middle of the zooming window. Therefore, the zoom factor is adjusted to display one pixel more than the power of 2 indicated by the zooming factor. The width and height of the zoomed part of the image hence are $2^{8-f}+1$, where f is the zooming factor.

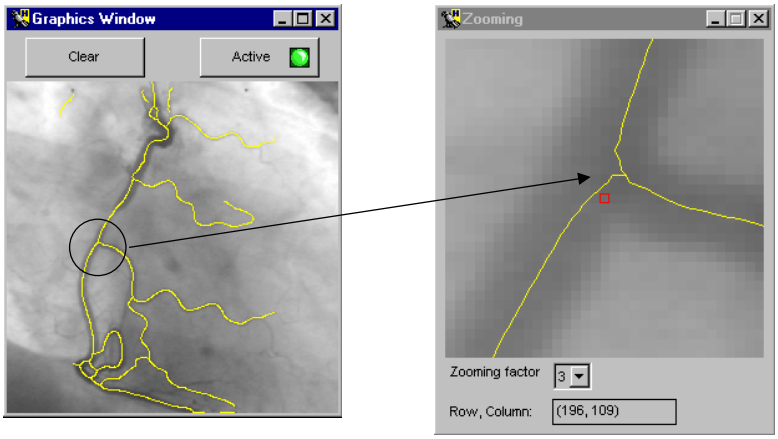

Figure 2.18: Real-time zooming.

You can select a particular pixel by single-clicking on it with the left mouse button. The zooming tool stores this position internally, and will redisplay the thus selected part of the image object when you leave the graphics window. This enables you to have a meaningful display in the zooming tool whenever you want to do actions outside of the graphics window.

2.3.6.7 `Visualization` ▷ `Gray Histogram Info`

This menu item opens a sophisticated tool for the inspection of gray value histograms, which can also be used to select thresholds interactively and to set the range of displayed gray values dynamically. Figure 2.19 shows the layout of the gray histogram inspection window.

When opening the tool, the histogram of the image shown in the currently active graphics window is displayed. When the tool is already open, four modes of sending new image data to the tool are available:

1. The simplest mode is to display an image in the active graphics window. Whenever you do so, the histogram of this image is computed and drawn, and the tool records the graphics window from which the image was sent.
2. Another simple method to send new data to the tool is to single-click into an image that is displayed in a graphics window.
3. Whenever image data is displayed overlaid with region data in a graphics window (the graphics window does not need to be active for this), you can click into any of the segmented regions, and the histogram of the image within that region will be computed and shown. If you click into a part of the image that is not contained in any of the overlaid regions, the histogram of the entire image will be displayed.
4. The same mechanism is used for regions that have gray value information, e.g., image objects created by `reduce_domain` or `add_channels`. Here, the histogram of the image object you click into will be displayed.

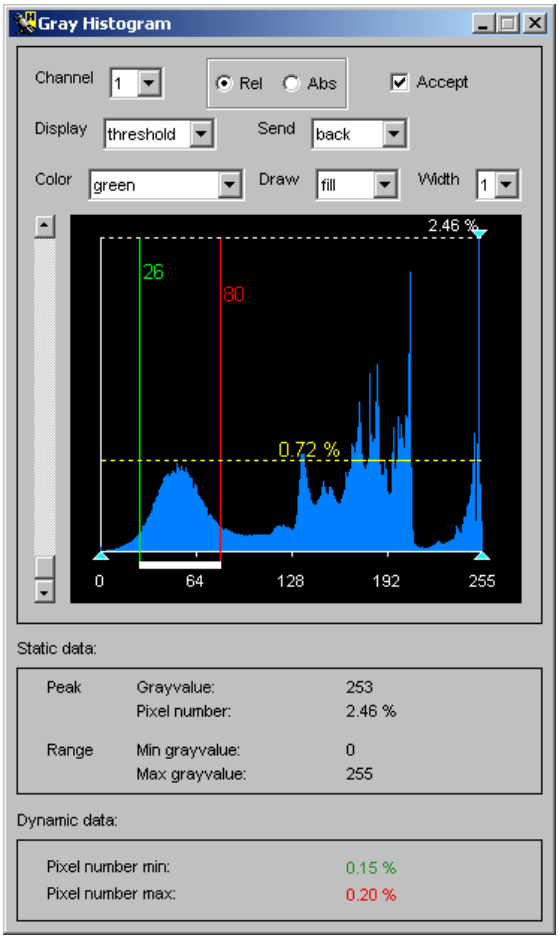

Figure 2.19: Online gray histogram inspection.

Note that when the graphics window the image was sent from is cleared, the histogram is not reset.

When a multi-channel image, e.g., a RGB color image, is sent to the tool, by default the histogram of the first channel is displayed. The combo box `Channel` lets you select the channel from which to compute the histogram.

The radio box in the upper center of the tool lets you select whether to display the histogram with relative or absolute frequencies. When you select `Rel`, the frequencies of individual gray values are displayed as percentages; when you select `Abs`, they are displayed as the actual number of pixels having a particular gray value. See the operator `gray_histo` in the HALCON Reference Manuals for details.

Sometimes, it is desirable to suppress the updating of the histogram when new image data is available, e.g., if you want to select thresholds for a gradient image, but want to visualize the original image along with the segmentation (see below). To do so, you can deselect the checkbox `Accept`.

The main part of the tool is the area, in which the histogram of the image is displayed in blue. This area

contains static parts and parts that can be interactively manipulated. The first static part is the horizontal coordinate axis, which displays the gray values in the image. For byte images, this range is always 0 ... 255. For all other image types, e.g., real images, the horizontal axis runs from the minimum to the maximum gray value of the image, and the labeling of the axis is changed accordingly. To the left of the display, the vertical axis representing the frequency of the gray values is drawn in white. The final static parts of the display are three cyan arrows. The two upward pointing arrows denote the maximum and minimum gray value of the image. The downward pointing arrow denotes the gray value that occurs most frequently, i.e., the peak of the histogram. These data are displayed in textual form within the `Static data` area of the display.

The dynamic parts of the histogram area are the three colored lines, which can be manipulated. The dashed horizontal yellow line can be dragged vertically. The label on this line indicates the frequency of gray values below this line. It can be used to interactively measure the frequency of gray values at different parts of the histogram. The vertical green and red lines denote the minimum and maximum selected gray value of the histogram, respectively. The selected range is drawn as a white bar below the horizontal gray value axis. The gray values on which the two vertical lines lie are displayed next to the lines in the same color. The frequency of the respective gray values is displayed within the `Dynamic data` area of the display. Initially, the histogram is displayed at full vertical range, i.e., up to the peak value, denoted by the label on the dashed white line. The scrollbar to the left allows to scale the histogram vertically. The label on the dashed white line will be updated accordingly.

The selected range of gray values can be used for two major purposes: thresholding (segmentation) and scaling the gray values:

Automatic segmentation (thresholding)

If the combo box `Display` is set to `threshold`, the image from which the histogram was computed is segmented with a `threshold` operation with the selected minimum and maximum gray value. Depending on the setting of the combo box `Send`, the segmentation result is either displayed in the graphics window, from which the image was originally sent (`Send = back`), or in the active graphics window (`Send = to active`). With the three combo boxes `Color`, `Draw`, and `Width`, you can specify how the segmentation results are displayed (see also `Visualization` ▷ `Color`, `Visualization` ▷ `Draw`, and `Visualization` ▷ `Line Width` below).

If you want to select threshold parameters for a single image, display the image in the active graphics window and open the histogram tool. For optimum visualization of the segmentation results, it is best to set the visualization color to a color different from black or white. Now, set `Display` to `threshold` and interactively drag the two vertical bars until you achieve the desired segmentation result. The parameters of the threshold operation can now be read off the two vertical lines.

If you want to select threshold parameters for an image which is derived from another image, but want to display the segmentation on the original image, e.g., if you want to select thresholds for a gradient image, two different possibilities exist. First, you can display the derived image, open the histogram tool, deselect `Accept`, display the original image, and then select the appropriate thresholds. This way, only one window is needed for the visualization. For the second possibility you can display the derived image in one window, activate another window or open a new window, display the original image there, activate the first window again, open the histogram tool, activate the second window again, set `Send` to `to active`, and select your thresholds. Although in this case it is not necessary to deselect `Accept`, it is advantageous to do so, because this prevents the histogram from being updated if you click into a graphics window accidentally.

Scaling the gray values

If `Display` is set to `scale`, the gray values of the image are scaled such that the gray value 0 of the scaled image corresponds to the minimum selected gray value and the gray value 255 to the maximum selected gray value. Again, the combo box `Send` determines the graphics window, in which the result is displayed. This mode is useful to interactively set a "window" of gray values that should be displayed with a large dynamic range.

2.3.6.8 `Visualization` ▷ `Feature Histogram Info`

This menu item opens a sophisticated tool for the inspection of feature histograms. In contrast to the gray value histogram described above, this tool does not inspect individual pixels, but regions or XLDs; for these iconic objects, it displays the distribution of values of a selected *feature*, e.g., the area of an XLD or the mean gray value of the pixels within a region. The feature histogram can also be used to select suitable thresholds for the operators `select_shape` and `select_shape_xld` interactively.

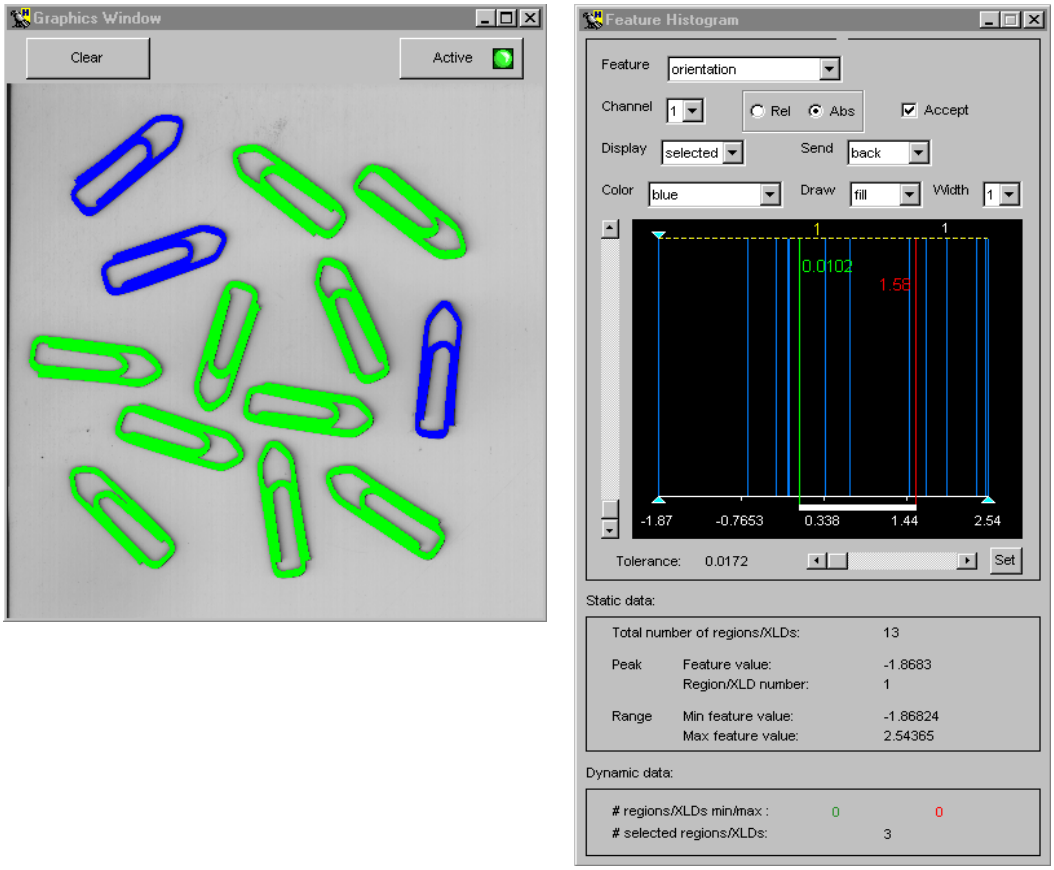

Figure 2.20: Selecting clips with a certain angle using the region feature histogram.

Figure 2.20 shows the feature histogram window together with the graphics window. Upon opening, the tool displays the histogram of the area (default feature selection) of the regions or XLDs that were displayed most recently in the currently active graphics window. You can select various features in the combo box `Feature`; the status bar displays additional information about the select feature, e.g., which operator is used to calculate it. Further information about region features can be found in the description of the menu item `Visualization` ▷ `Region Info`.

Most parts of the tool are built up similarly to the gray value histogram, which is described in detail above (menu entry `Visualization` ▷ `Gray Histogram Info`); in the following, we concentrate on points specific to the feature histogram. An important point regards the "source" of the regions or XLDs: The feature histogram is calculated for the regions or XLDs that were displayed most recently in the graphics window. Thus, if you display an image, there are no regions or XLDs, therefore the histogram remains "empty". As soon as you display regions or XLDs on top of an image, the histogram is calculated. If you display regions or XLDs without an image, you can still calculate feature histograms, but only for shape features. Please keep in mind that only the most recently displayed regions or XLDs are the source of the histogram, not all objects currently displayed in the graphics window!

The histogram itself is displayed with the horizontal axis corresponding to the feature values and the vertical axis corresponding to the frequency of the values, i.e., to the number of regions or XLDs with a certain feature value. In figure 2.20, the histogram of the orientation of the clips is displayed. When comparing the histogram to the gray value histogram in figure 2.19 on page 34, you will note a typical difference: Because in most cases the overall number of regions or XLDs is much smaller than the overall number of pixels, feature histograms often consist of individual lines, most of them having the height 1 (when displaying absolute frequencies). Of course, this effect depends on the selected feature: For features with floating-point values, e.g., the orientation, the probability that two regions or XLDs have the same feature value is very small, in contrast to features with integer values, e.g., the number of holes.

If you click on the button `Set`, a dialog that lets you modify the displayed range of feature values appears.

You can influence the calculation of the histogram with the slider `Tolerance`. The selected value is used to discretize the horizontal axis: Instead of determining the frequency of an "exact" feature value, regions with feature values falling within discrete intervals are summed. Graphically speaking, the horizontal axis is subdivided into "bins" with a width equal to the value selected with the slider `Tolerance`.

Automatic selection

As already noted, the region feature histogram facilitates the task of finding suitable threshold parameters for the operators `select_shape`, `select_gray`, and `select_shape_xld`: Select the entry `selected` in the combo box `Display`, choose suitable visualization parameters in the three combo boxes `Color`, `Draw`, and `Width`, and then position the two vertical lines such that the desired regions are highlighted. For example, in figure 2.20 the three clips pointing upwards and to the right were selected. The values displayed next to the vertical lines can then directly be used for the parameters `Min` and `Max`, the feature name selected in the combo box `Feature` for the parameter `Features`.

2.3.6.9 `Visualization` ▷ `Region Info`

This menu item opens a tool for the convenient inspection of shape and gray value features of individual regions. It can, for instance, be used to determine thresholds for operators that select regions based on

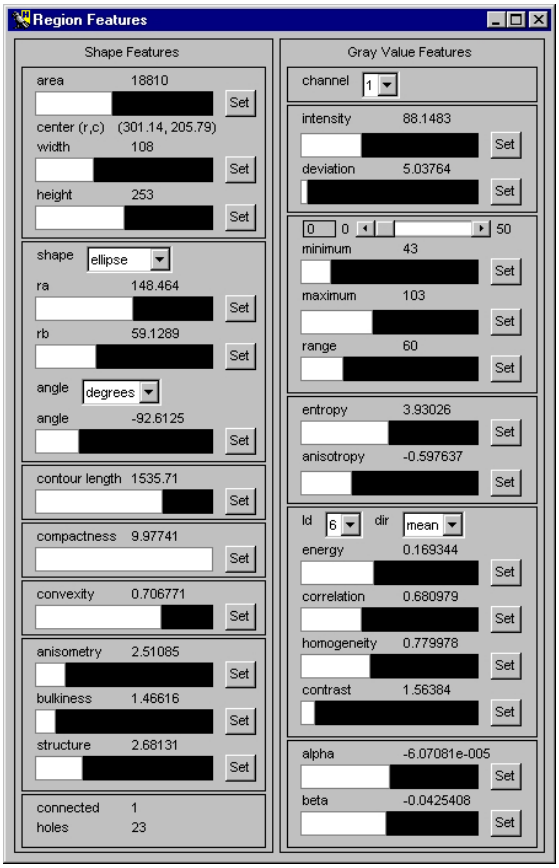

Figure 2.21: Online region feature inspection.

these features, e.g., `select_shape` or `select_gray`. Figure 2.21 shows the layout of the region feature inspection window.

The strategy to determine the data from which to compute the features is very similar to that of the gray histogram inspection window (see section 2.3.6.7 on page 33). You can display an image or region by double-clicking on it in the variable window or you can select a region or an image which is already displayed by single-clicking it. If you display or click into an image, the gray value features of the entire image will be calculated. If you click into a region that is not underlaid with an image, only the shape features of this region will be displayed. If you click into a region that is underlaid with an image or into a region that has gray value information (e.g., from `reduce_domain` or `add_channels`), both the shape and gray value features of that region will be displayed. Finally, if you have overlaid an image with a region, but click into a part of the image that is outside the region, only the gray value features of the entire image will be calculated.

Analogously to the gray histogram inspection window, the gray value features of a multi-channel image are calculated from the first channel by default. You can use the combo box `Channel` to select the desired channel.

Shape features

The shape features on the left side of the region inspection window are grouped into seven categories, which correspond roughly to individual HALCON shape feature operators. The top-most of the displays shows the most basic region features, namely the area and center of gravity of the region (see area_center in the Reference Manual) and the width and height of the smallest axis-parallel rectangle of the region. The latter is computed from the output of the operator smallest_rectangle1.

The second display contains information about the orientation (angle) and size of the region along the two principal directions (ra and rb) of the region. With the combo box shape, you can select by what means the size is computed. If you select ellipse, the size is computed with the operator elliptic_axis. This means that the parameters ra and rb are the major and minor axis of an ellipse that has the same moments as the selected region. Note that this ellipse need not enclose the region. If you set shape to rectangle, the size is computed with the operator smallest_rectangle2. This means, that ra and rb are half the width and height of the smallest rectangle with arbitrary orientation that completely contains the selected region. The orientation of the region is computed in both cases with the operator orientation_region to get the full range of 360° for the angle. You can select whether to display the angle in degrees or radians with the corresponding combo box.

The next three displays show simpler shape features of the selected region. The first of these displays shows the contour length of the region, i.e., the euclidean length of its boundary (see the operator contlength). The second one shows the compactness of the region, i.e., the ratio of the contour length of the region and the circumference of a circle with the same area as the region (see the operator compactness). The compactness of a region is always larger than 1. The compacter the region, the closer the value of the compactness is to 1. The third display shows the convexity of the selected region, i.e., the ratio of the area of the region and the area of the convex hull of the region (see the operator convexity). The convexity of a region is always smaller than 1. Only convex regions will reach the optimum convexity of 1.

The last but one display shows shape features derived from the ellipse parameters of the selected region, which are calculated with the operator eccentricity. The anisometry of the region is the ratio of the major and minor axis of the ellipse (i.e., the ratio of ra and rb in the second display if you set shape to ellipse). This feature measures how elongated the region is. Its value is always larger than 1, with isometric regions having a value of 1. The definition of the more complex features bulkiness and structure factor (abbreviated as structure in the display) can be obtained from the HALCON Reference Manual.

The final shape feature display shows the connected components and number of holes of the selected region, as computed by the operator connect_and_holes.

Gray value features

The gray value features are grouped into five displays on the right side of the region inspection window. Again, they correspond roughly to individual HALCON operators. The first display shows the mean gray value intensity and the corresponding standard deviation of the selected region. These are computed with the operator intensity.

The second display shows the output of the operator min_max_gray. This operator computes the distribution (histogram) of gray values in the image and returns the gray values corresponding to an upper and lower percentile of the distribution. This percentile can be selected with the slider at the top of the

display. For a percentile of 0 (the default), the minimum and maximum gray values of the region are returned. The display also shows the range of gray values in the region, i.e., the difference between the maximum and minimum gray values.

In the third display, the gray value entropy of the selected region is displayed (see the operator `entropy_gray`). Again, this is a feature derived from the histogram of gray values in the region. The feature entropy measures whether the gray values are distributed equally within the region. This measure is always smaller than 8 (for byte images — the only supported image type for this operator). Only images with equally distributed gray values reach this maximum value. The feature `anisometry` measures the symmetry of the distribution (see the operator `anisometry`. Perfectly symmetric histograms will have an anisometry of -0.5.

The fourth display contains gray value features derived from the coocurrence matrix of the selected region are displayed (see the operator `cooc_feature_image`). The combo box `ld` can be used to select the number of gray values to be distinguished (2^{ld}). The combo box `dir` selects the direction in which the coocurrence matrix is computed. The resulting features — energy, correlation, homogeneity, and contrast — have self-explanatory names. A detailed description can be found in the reference of the operator `cooc_feature_matrix`.

The final display contains the output of the operator `moments_gray_plane`. These are the angles of the normal vector of a plane fit through the gray values of the selected region.

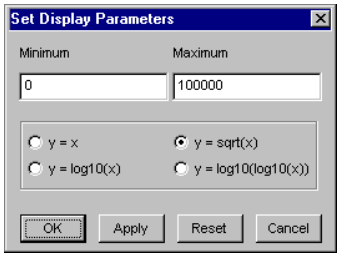

Figure 2.22: Configuration dialog for single region features.

Each of the gauges corresponding to a display can be configured to set the minimum and maximum values for each gauge. Furthermore, the scaling function of the gauge can be determined. This can be used to make the relation of the features of different regions more intuitive. For example, features that depend on the area of the region are more intuitively grasped when the scaling is set to `sqrt(x)`. The configuration dialog is the same for all gauges, and is shown in figure 2.22. It can be brought up by pressing the button next to each gauge.

2.3.6.10 `Visualization` ▷ `Size Window`

There are convenient methods to change the size of the active graphics window dependent on the size of the previously displayed image. Using the submenu `Original`, the window is set to the same size as the most recently displayed image, that means, for each pixel of the image one pixel on the screen is used for displaying. Similar to this, you can select `Original half` or `Original quarter` to make the window half or a quarter as big as the displayed image. The submenus `Half` and `Double` change the size

of the graphics window to half and double its current size, respectively, independent of the size of the previously displayed image. You can combine `Double` with `Original`. The submenu `Aspect` changes the aspect ratio of the graphics window, so that pixels are displayed as squares on the screen. For this operation, again the size of the previously displayed image is used.

2.3.6.11 `Visualization` ▷ `Zooming`

This is a convenient menu for manipulation of the zooming mode. The submenu `Reset` switches zooming off, i.e., an image will be displayed so that it fills the graphics window completely. The submenus `Zoom In` and `Zoom Out` apply a zooming "in" and "out" to the image or region by a factor of two. Finally, there are two interactive modes to control zooming: `Draw Rectangle` allows the specification of a rectangular part of the window to be zoomed while `Draw Center` allows the definition of a pixel coordinate that should be at the center of the window (e.g., for a successive `Zoom In`).

For more information see the menu `Visualization` ▷ `Set Parameters` ▷ `Zoom`.

2.3.6.12 `Visualization` ▷ `Colored`

This is an easy way to display multiple regions or XLDs. Each region is displayed in a different color, where the number of different colors is specified in the submenu. You can choose between 3, 6 and 12 colors. If all regions are displayed with one color, you have to use the operator `connection` beforehand. You can check this also with the operator `count_obj`.

2.3.6.13 `Visualization` ▷ `Color`

This item allows you to choose a color for displaying segmentation results (regions and XLDs), text (`write_string`) and general line drawings (e.g., 3D plots, contour lines, and bar charts). The number of colors which are available in the submenu depends on the graphics display (i.e., the number of bits used for displaying). After selecting a color, the previously displayed region or XLD object will be redisplayed with this color. The default color is white.

2.3.6.14 `Visualization` ▷ `Draw`

Here, you can select a visualization mode to display regions. It can either be *filled* (item `fill`) or the *borders* are displayed only (item `margin`). The border line thickness of the displayed regions is specified using the menu item `Line Width` (see figure 2.24 on page 44).

2.3.6.15 `Visualization` ▷ `Line Width`

Here, you determine the line width for painting XLDs, borders of regions or other types of lines. You can select between a wide range of widths using the submenu.

2.3.6.16 Visualization ▷ Shape

Here, you specify the representation shape for regions. Thus you are able to display not only the region's original shape but also its enclosing rectangle or its enclosing circle.

2.3.6.17 Visualization ▷ Lut

This menu item activates different look-up tables, which can be used to display gray value images and color images in different intensities and colors. In the case of a true color display the image has to be redisplayed due to the missing support of a look-up-table in the graphics hardware. For color images only the gray look-up-tables can be used, which change each channel (separately) with the same table.

2.3.6.18 Visualization ▷ Paint

This menu item defines the mode to display gray value images. For more information see the menu item Visualization ▷ Set Parameters below.

2.3.6.19 Visualization ▷ Set Parameters

By using this menu item, a dialog called Visualization Parameters is opened, which handles more complex parameter settings. Select one setting with your left mouse button and the window brings up the according parameter box. Each box contains different buttons, text fields, or check boxes to modify parameters.

Each box has an Update button. If this button is pressed, every change of a parameter will immediately lead to a redisplay of the image, regions, or XLD in the graphics window. If the button is "off" the parameters become active for the next display of an object (double click on an icon or execution of an operator). By default the update is deactivated for the boxes Lut and Paint.

You may specify the following parameter settings.

Paint
: Here, you can select between several graphical presentations for images. Examples are contourline and 3D-plot. In the default mode the image will be displayed as a picture (see figure 2.23).

  If you have chosen a presentation mode, the window displays all possible parameters you may modify. For example, after selecting the item 3D-plot you have to specify the following parameters:

  - Step (the distance of plot lines in pixels),
  - Colored (use the gray value of a pixel to draw a line segment instead of one graphic color),
  - Eye height,
  - Eye distance (view point),
  - Scale (height of 3D plot),

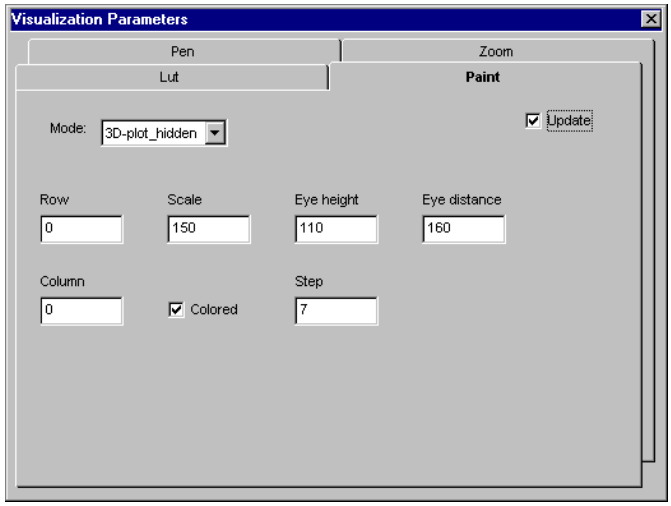

Figure 2.23: Settings of parameter `paint`.

- Row and
- Column (position of the center).

`Pen`

Here, the display modes display for regions and XLDs are specified. You can select the color (single or multiple), the drawing mode (filled or border), the line width, and the shape of the regions.

You can select up to 12 **colors** by clicking the appropriate checkbox. They are used to emphasize the connectivity of different regions in the graphics window. If you choose a single color presentation you may specify this color by selecting it in the list box (see figure 2.24).

With the parameter **shape** (default is `original`), you may specify the presentation shape for regions. Thus you are able to display not only the region's original shape but also its enclosing rectangle or its enclosing circle, etc.

The **line width** of the presented regions, XLDs, or lines is specified with help of the menu item `border width`.

For regions the **draw mode** can be specified: Either it might be *filled* (item `fill`) or the *borders* are displayed (item `margin`) only.

`Zoom`

The menu item specifies which part of an image, region, XLD, or other graphic item is going to be displayed (see figure 2.25). The upper left four text fields specify the coordinate system. `left/upper` defines the pixel which will be displayed at the upper left corner of the window. `lower/right` defines the pixel which will be displayed at the lower right side of the window. By selecting the upper button `Interactive...` you specify a rectangular part in the graphics window interactively. For this, you press the left mouse button to indicate the rectangle's upper left corner. Hold the button and drag the mouse to the lower right corner's position. Release the

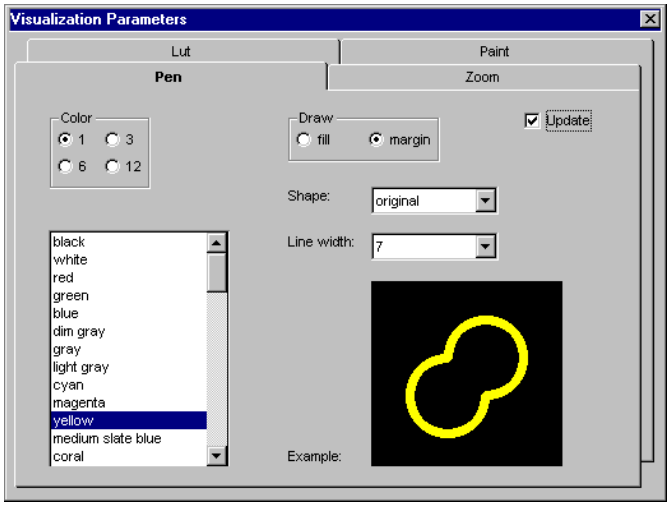

Figure 2.24: Settings of parameter `pen`.

button and correct the size by grabbing the borders or corners of the rectangle. By pressing the right mouse button inside your specified rectangle you display the objects inside the rectangle in the graphics window.

You can also enter the coordinates of the desired clipping manually by specifying the coordinates of the upper left corner and the lower right corner in the respective text fields. Please note that the text fields sometimes behave unexpectedly.

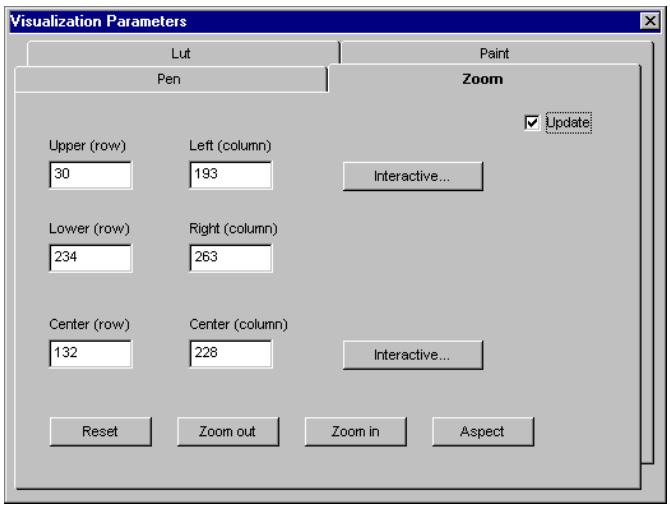

Figure 2.25: Settings of parameter `zoom`.

Below the coordinates of the rectangle, you see its center. This center can also be specified inter-

actively using the button `Interactive...`. Activating this mode, you first have to click with the left button into the active graphics window. Now you can correct this position by again pressing the left mouse button. To quit, press the right mouse button.

The buttons `Zoom Out` and `Zoom In` activate a zooming with factor 2 or 0.5, respectively.

To get the image's full view back on your graphics window you simply click the checkbox `Reset`.

`Lut`

Using `Lut` you are able to load different look-up-tables for visualization (see figure 2.26). With the help of a false color presentation you often get a better impression of the gray values of an image. In the case of a true color display, the image has to be redisplayed due to the missing support of a look-up-table in the graphics hardware. For color images only the gray look-up-tables can be used, which change each channel (separately) with the same table.

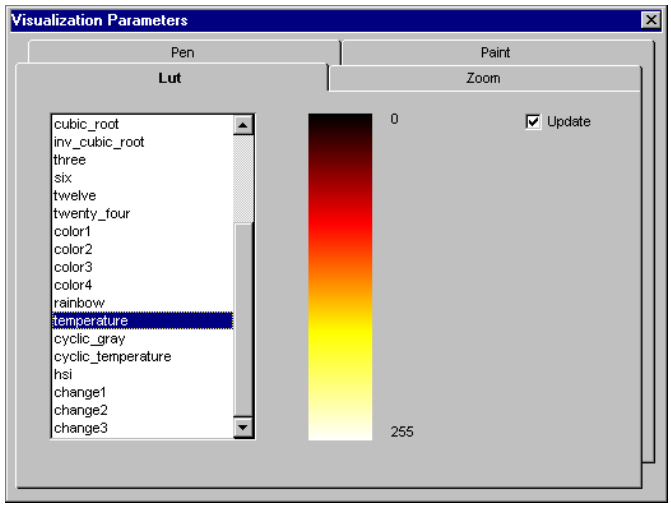

Figure 2.26: Settings of parameter `lut`.

2.3.7 Menu `Procedures`

The menu procedures contains all functionality that is needed to create, modify, copy, delete, or save HDevelop procedures. All procedures in the current program except the main procedure are appended in alphabetical order at the bottom of this menu. The menu items that can be selected are shown in figure 2.27.

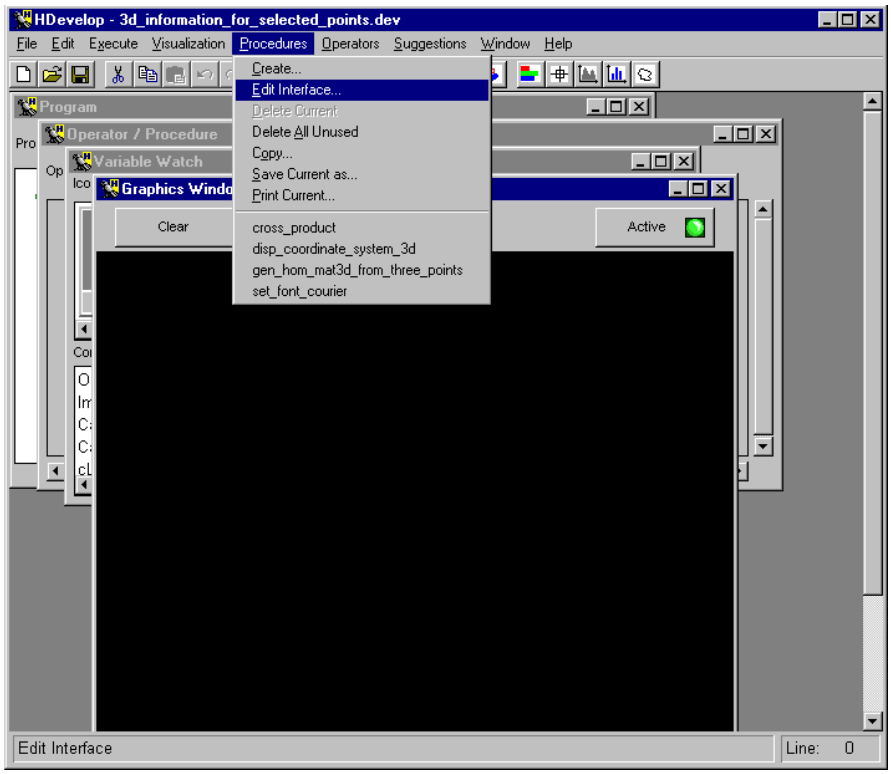

Figure 2.27: The menu `Procedures`.

2.3.7.1 `Procedures` ▷ `Create`

This menu item is used to create a new HDevelop procedure and corresponds to the button `Create` in the program window. Selecting this item opens the procedure interface dialog, which, if necessary, replaces the operator dialog, i.e., only one of the two dialogs can be open at a time. The procedure interface dialog and the mechanism of creating procedures are described in section 2.4.3 on page 65.

2.3.7.2 `Procedures` ▷ `Edit Interface`

This menu item opens the procedure interface dialog and displays the interface of the current procedure. `Procedures` ▷ `Edit Interface` has no effect if the current procedure is the main procedure. The menu

item has the same effect as the button `Interface` in the program window (compare section 2.4.3 on page 65).

2.3.7.3 `Procedures` ▷ `Delete Current`

The current procedure is deleted from the program and the main procedure becomes the current procedure. All calls to the procedure in the current program are replaced by comments. This item is disabled if the current procedure has a procedure call on the call stack or is the main procedure.

2.3.7.4 `Procedures` ▷ `Delete All Unused`

All procedures that cannot be reached by any procedure call from the main procedure are deleted from the program. If the current procedure is among the deleted procedures, the main procedure becomes the current procedure.

2.3.7.5 `Procedures` ▷ `Copy`

Selecting this menu item opens a dialog with which it is possible to copy existing procedures (see figure 2.28). The dialog offers a combo box in which you can select the procedure that is to be copied, either by typing the procedure name in the text field or by selecting a procedure from the combo box list, which contains all procedures in the program. In the target text field the name of the target procedure can be entered. Activating the `Copy` button creates a copy of the source procedure and adds it to the current program. `Cancel` dismisses the dialog.

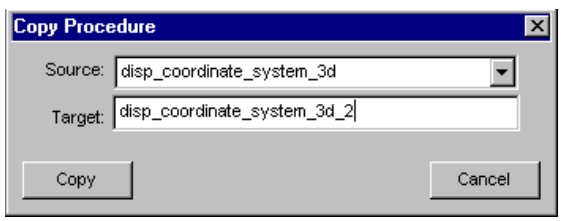

Figure 2.28: Copy procedures.

2.3.7.6 `Procedures` ▷ `Save Current As`

HDevelop allows you to save or export the current procedure to a file. A file selection dialog similar to to the one of `File` ▷ `SaveAs` (see section 2.3.3.6 on page 18) is opened and after selecting a file, the current procedure is saved together with all procedures that can be called directly or indirectly from the procedure. If the selected file format is of type HDevelop, an empty main procedure is added to the file. Note that if the selected file format corresponds to one of the supported HDevelop export languages C, C++, or VB, the exported file will not contain a main or similar function, and the exported file will not work as a standalone application. If the current procedure is the main procedure `Procedures` ▷ `Save Current As` has the same effect as `File` ▷ `Save As`.

2.3.7.7 Procedures ▷ Print Current

The menu item Procedures ▷ Print Current enables you to print the current procedure's program body. Upon selecting the menu item, a dialog appears in which you can configure the printing process.

2.3.8 Menu Operators

This menu item comprises all HALCON and HDevelop operators including the HDevelop control constructs.

2.3.8.1 Operators ▷ Control

Here, you may select control structures for the program. This involves the execution of a program segment (henceforth named body) depending on a test (if and ifelse) and the repetition of a program segment (for, while, and break). Furthermore, you may stop the program's execution at any position (stop) or terminate HDevelop (exit). The operators assign and insert do not influence the execution, but serve to specify values for control data (assignment). The operator comment is used to add a comment, that means any sequence of characters, to the program. The operator return terminates the current procedure call and returns to the calling procedure (see section 2.2 on page 12 for more information about HDevelop procedures).

The corresponding menu is shown in figure 2.29.

Selecting a menu item displays the corresponding control construct in the operator window, where you can set the necessary parameters. After specifying all parameters you may transfer the construct into your program. A direct execution for loops and conditions is not possible, in contrast to other HDevelop and HALCON operators, because you have to specify the loop's and condition's body first to obtain useful semantics. If necessary, you may execute the program after the input with Step or Run. The insertion cursor is positioned after the construct head to ensure the input of the construct's body occurs in the correct place. This body is indented to make the nesting level of the control constructs visible, and thus to help you in understanding the program structure (see figure 2.30). To get an idea how to use loops, you may look at the example session in section 1.3 on page 2, and at the programs in section 5.3 on page 124, section 5.8 on page 135 and section 5.9 on page 137. The semantics for loops and conditions are shown in section 3.7 on page 100.

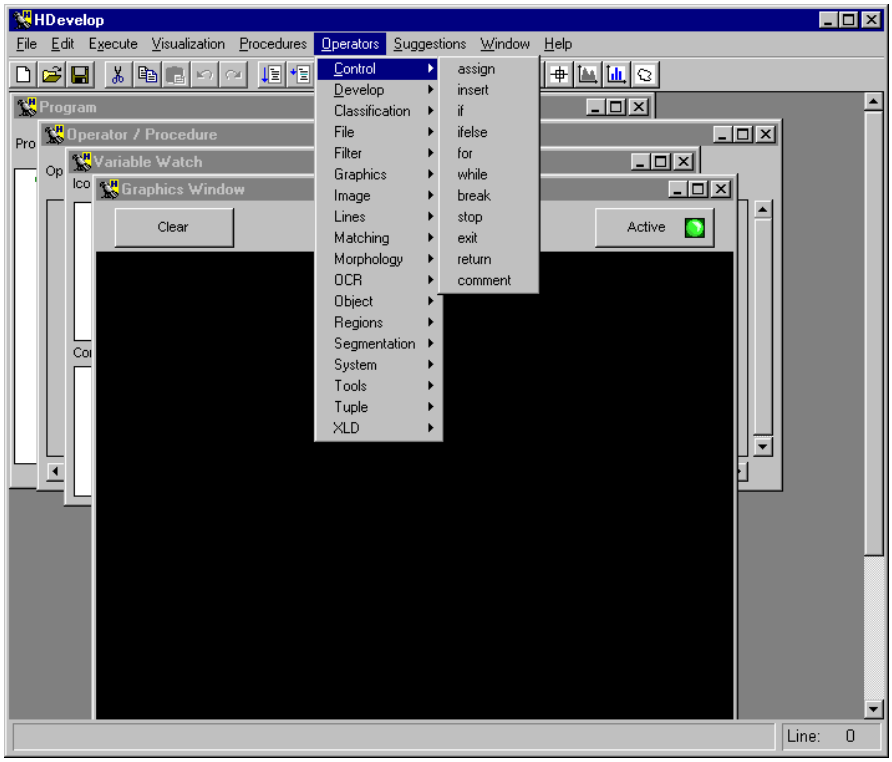

Figure 2.29: Menu item `Operators` ▷ `Control`.

Assignment

The operator `assign` serves as an assignment operator for control variables (numbers and strings). Analogously to "normal" operators the input is made in the operator window by specifying both "parameters" `Input` and `Result` (i.e., right and left side of the assignment). An instruction in C, e.g.,

```
x = y + z;
```

is declared inside the operator window as

```
assign(y + z,x)
```

and displayed in the program window by

```
x := y + z
```

The operator `insert` implements the assignment of a single value (tuple of length 1) at a specified index position of a tuple. Thus an array assignment (here in C syntax)

```
a[i] = v;
```

is entered as

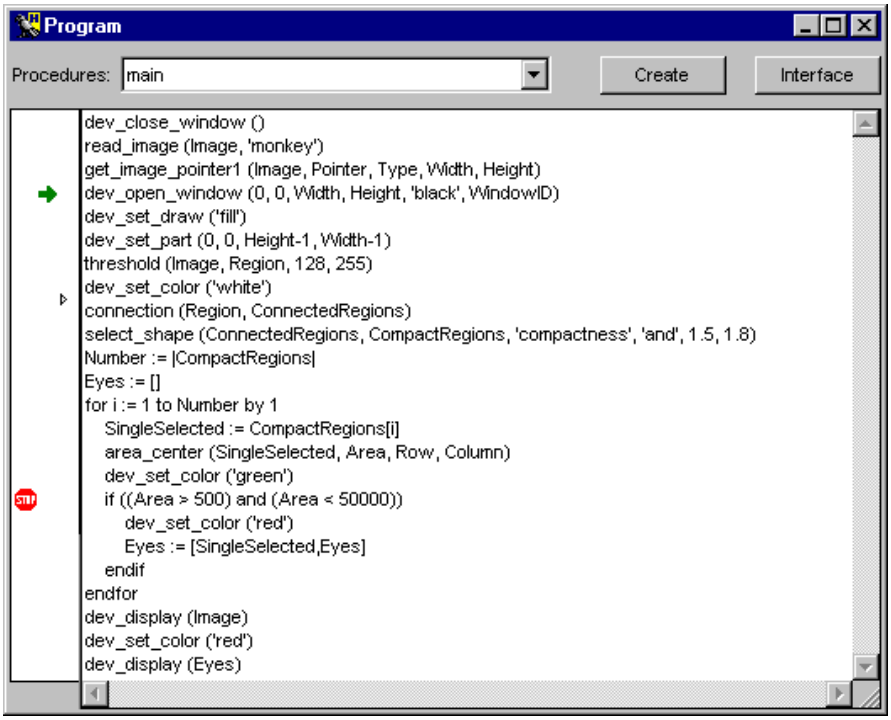

Figure 2.30: Example for using a `for` loop.

```
insert(a,v,i,a)
```

in the operator window, and is displayed as

```
a[i] := v
```

in the HDevelop program window.

Program termination

The operators `stop` and `exit` are used to terminate the program. More precisely, `stop` *interrupts* an execution and `exit` *terminates* HDevelop. Having interrupted the execution you may continue the program by pressing `Step` or `Run`. This is useful, e.g., in demo programs to install defined positions for program interruption. Under UNIX, you can use `exit` in combination with a startup file and the option `-run` (see section 1.3 on page 2). Thus, HDevelop will not only load and run your application automatically, but also terminate when reaching `exit`.

Comments

The operator `comment` allows to add a line of text to the program. This text has no effect on the execution of the program. A comment may contain any sequence of characters.

2.3.8.2 Operators ▷ Develop

This menu contains several operators that help to adapt the user interface. These operators offer the same functionality that you have using mouse interaction otherwise. They are used to configure the environment from within a program. Using these operators, the program performs actions similar to the setting of a color in the parameter window, opening a window in the menu bar, or iconifying the program window with the help of the window manager.

All operators in this menu start with the prefix `dev_`. It has been introduced to have a distinction to the underlying basic HALCON operators (e.g., `dev_set_color` and `set_color`). You can find the complete listing in figure 2.31.

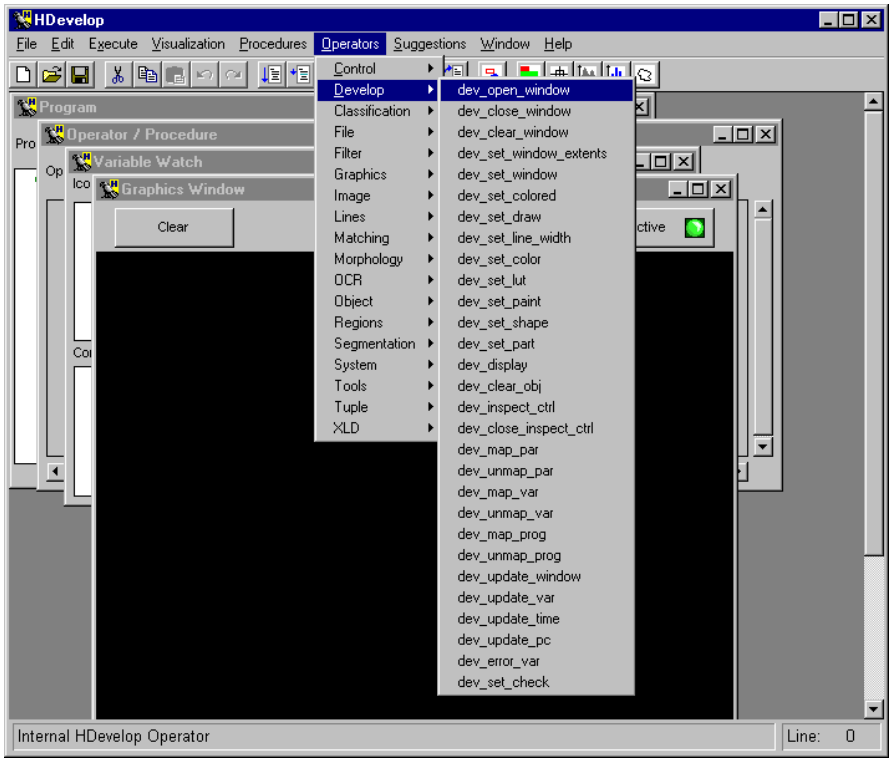

Figure 2.31: Operators in menu item `Operators` ▷ `Develop`.

The effects of each operator are described as follows:

- `dev_open_window`, `dev_close_window`, `dev_clear_window`
  The operators `dev_open_window` and `dev_close_window` are used to open and to close a graphics window, respectively. During opening, the parameterization allows you to specify the window's size and position. The operator `dev_clear_window` clears the active window's content and its history. This corresponds to the usage of the button `Clear` in the graphics window. Please note that `dev_open_window` and `dev_close_window` are not supported for Visual Basic export because here one `HWindowXCtrl` is used.

- dev_set_window_extents
  With this operator, you can set the size and position of the active HDevelop graphics window.
- dev_set_window
  This operator activates the graphics window containing the given ID. This ID is an output parameter of dev_open_window. After the execution, the output is redirected to this window. This operator is not needed for exported code in C++, because here every window operation uses the ID as a parameter. The operator has no effect for exported code in Visual Basic.
- dev_set_color, dev_set_colored
  dev_set_color has the same effects as the menu item Visualization ▷ Color (see section 2.3.6.13 on page 41). dev_set_colored is equal to the menu item Visualization ▷ Colored (see section 2.3.6.13 on page 41).
- dev_set_draw
  This operator has the same effects as the menu item Visualization ▷ Draw (see section 2.3.6.14 on page 41).
- dev_set_line_width
  For an explanation see the menu item Visualization ▷ Line Width (see section 2.3.6.15 on page 41).
- dev_set_lut
  For an explanation see the menu item Visualization ▷ Lut (see section 2.3.6.17 on page 42).
- dev_set_paint
  For an explanation see the menu item Visualization ▷ Paint (see section 2.3.6.18 on page 42). If you want to specify all possible parameters of a given paint mode, you have to specify them as a tuple, analogously to the HALCON operator set_paint.
- dev_set_shape
  For an explanation see the menu item Visualization ▷ Shape (see section 2.3.6.16 on page 42).
- dev_set_part
  This operator adjusts the coordinate system for image, region, XLD and other graphic output. This is done by specifying the upper left and the lower right corner coordinates. This specified part is shown in the entire graphics window. If the width or height of the specified rectangle has a negative value (e.g., Row1 > Row2) the result is equivalent to the menu Visualization ▷ Zooming ▷ Reset (see section 2.3.6.11 on page 41): the zoom mode is switched off, i.e., the *most recently* displayed image fills the whole graphics window. This feature of dev_set_part is *not* supported for exported C++ and Visual Basic code.
- dev_display
  Iconic variables are displayed in the active graphics window by this operator. It is reasonable to do this when the automatic output is suppressed (see dev_update_window and File ▷ Options (see section 2.3.3.11 on page 20)).
- dev_clear_obj
  This operator deletes the iconic object stored in the HDevelop variable that is passed as the input parameter. In the variable window, the object is displayed as undefined (with a ? as its icon).
- dev_inspect_ctrl

This operator opens an inspection window displaying the values of the variable passed to the operator. In most cases a list dialog is opened, which shows *all* values of the variable. In the case of a frame grabber handle, a description of this frame grabber is opened. In addition, this dialog allows online grabbing of images. This operator is not supported for exported C++ and Visual Basic code.

- dev_close_inspect_ctrl
  This is the opposite operator to dev_inspect_ctrl, and closes the inspect window. This operator is not supported for exported C++ and Visual Basic code.

- dev_map_par, dev_unmap_par
  These operators open and close the parameter dialog, which can also be opened using the menu Visualization ▷ Set Parameters (see section 2.3.6.19 on page 42). This operator is not supported for exported C++ and Visual Basic code.

- dev_map_var, dev_unmap_var
  These operators iconify the variable window (dev_unmap_var), and retransform the iconified window to the normal visualization size, respectively (dev_map_var). This means that the variable window always remains visible on the display in one of the two ways of visualization. These operators can be executed with the help of the window manager. These operators are not supported for exported C++ and Visual Basic code.

- dev_map_prog, dev_unmap_prog
  Analogously to dev_map_var and dev_unmap_var, these operators iconifyiconify or deiconify the program window. These operators are not supported for exported C++ and Visual Basic code.

- dev_update_window, dev_update_var, dev_update_time, dev_update_pc
  Using these operators, you may configure the output at runtime. It corresponds to the settings in menu File ▷ Options (see section 2.3.3.11 on page 20). These operators are not supported for exported C++ and Visual Basic code.

- dev_set_check
  This operator is equivalent to set_check of the HALCON library. It is used to handle runtime errors caused by HALCON operators that are executed inside HDevelop. The parameter value 'give_error', which is the default, leads to a stop of the program together with an error dialog if a value not equal to H_MSG_TRUE is returned. Using the value '~give_error', errors or other messages are ignored and the program can continue. This mode is useful in connection with operators like get_mposition, file_exists, read_image, or test_region_point, which can return H_MSG_FAIL. An example can be found in section 5.10 on page 138.

- dev_error_var
  This operator specifies a variable that contains the return value (error code) of an operator after execution. This value can be used to continue, depending on the given value. dev_error_var is normally used in connection with dev_set_check. Note that, as the procedure concept in HDevelop only allows for local variables, the variable set by dev_error_var will only be valid in calls to the relevant procedure. Furthermore, every corresponding procedure call will have an own instance of the variable, i.e. the variable might have different values in different procedure calls. For an example how to use dev_error_var in connection with dev_set_check see %HALCONROOT%\examples\hdevelop\Graphics\Mouse\get_mposition.dev.

Please note that operations concerning graphics windows and their corresponding operators have additional functionality as HALCON operators with corresponding names (without dev_): graphics windows

in HDevelop are based on HALCON windows (see `open_window` in the HALCON reference manual), but in fact, they have an enhanced functionality (e.g., history of displayed objects, interactive modification of size, and control buttons). This is also true for operators that modify visualization parameters (`dev_set_color`, `dev_set_draw`, etc.). For example, the new visualization parameter is registered in the parameter window when the operator has been executed. You can easily check this by opening the dialog `Visualization` ▷ `Set Parameters` ▷ `Pen` and apply the operator `dev_set_color`. Here you will see the change of the visualization parameters in the dialog box. You have to be aware of this difference if you export `dev_*` to C++ and Visual Basic code.

In contrast to the parameter dialog for changing display parameters like color, the corresponding operators (like `dev_set_color`) do not change to contents of the graphics window (i.e., they don't cause a redisplay). They are used to prepare the parameters for the *next* display action.

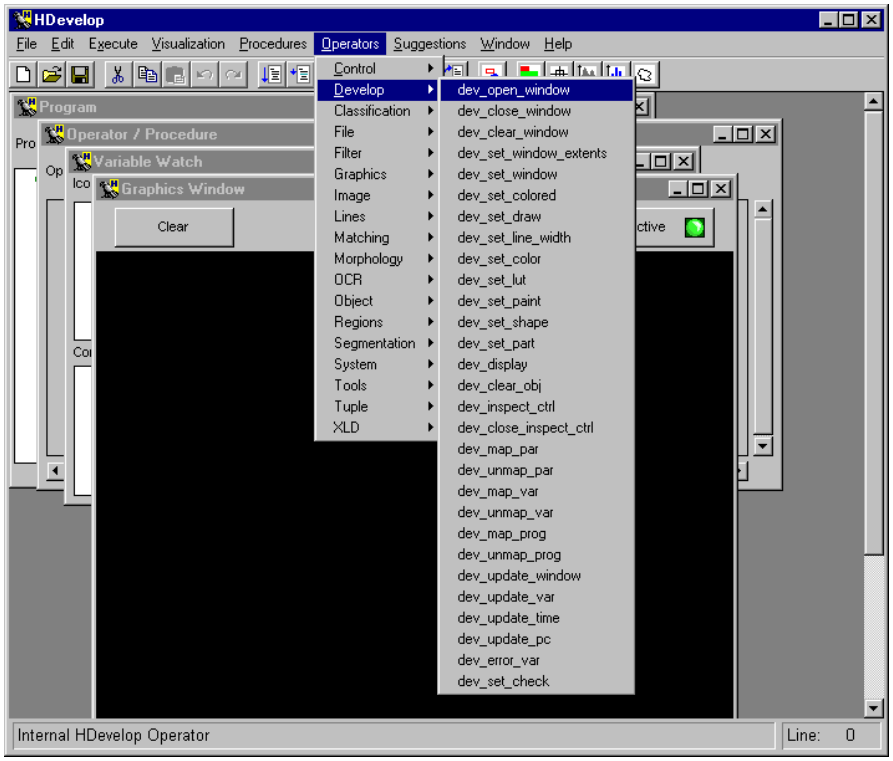

Figure 2.32: Menu hierarchy of all HALCON operators.

2.3.8.3 `Operators` ▷ `Classification, File, ...`

In the following items, you can find all HALCON operators, arranged in chapters and subchapters. This set of image analysis operators forms the most important part of HALCON: the HALCON library. HALCON operators implement the different image analysis tasks such as preprocessing, filtering, or measurement (see figure 2.32).

You may look for a detailed description of each operator in the HALCON reference manual.[3]

The menu has a cascade structure, according to the chapter structure of the HALCON reference manual. As this menu has to be built up after opening the program window, it might take some time until it is available. During the build-up time the menu is "grayed out". Selecting a chapter of the menu opens a pulldown menu with the corresponding subchapters or operators, respectively.

This operator hierarchy is especially useful for novices because it offers all operators sorted by thematic aspects. This might be interesting for an experienced user, too, if he wants to compare, e.g., different smoothing filters, because they reside in the same subchapter. To get additional information, a short description of an operator (while activating its name in the menu) is displayed in the status bar (see figure 2.32).

Note, that some operators are visible in the menus but cannot be selected, e.g., `open_window` (in `Operators` ▷ `Graphics` ▷ `Window`) or `reset_obj_db` (in `Operators` ▷ `System` ▷ `Database`) . In the case of most of these operators, you should use the corresponding `Develop` operator (e.g., `dev_open_window` instead of `open_window`) *within* HDevelop. Some operators, e.g. `reset_obj_db`, cannot be called at all within HDevelop.

[3] Operators of the menus `Control` and `Develop` are special operators of HDevelop. Thus you will not find them in the reference manuals for HALCON/C, HALCON/C++, or HALCON/COM.

2.3.9 Menu Suggestions

This menu shows you another possibility how to select HALCON operators. But here they are proposed to you in a different manner. It is assumed that you have already selected or executed an operator in a previous step. Depending on this operator, five different suggestions are offered. Figure 2.33 shows possible successor suggestions for operator read_image.

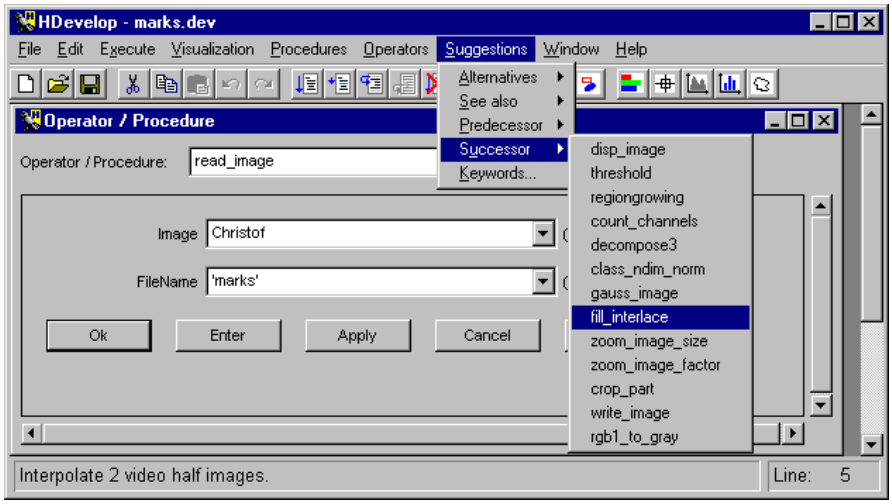

Figure 2.33: Suggestions to select a successor of HALCON operator read_image.

Suggestions are separated into groups as described below.

2.3.9.1 Suggestions ▷ Predecessor

Many operators require a reasonable or necessary predecessor operator. For example, before computing junction points in a skeleton (junctions_skeleton), you have to compute this skeleton itself (skeleton). To obtain a threshold image you have to use a lowpass filter before executing a dynamic threshold (dyn_threshold). Using the watershed algorithms (watersheds), it is reasonable to apply a smoothing filter on an image first, because this reduces runtime considerably.

2.3.9.2 Suggestions ▷ Successor

In many cases the task results in a “natural” sequence of operators. Thus as a rule you use a thresholding after executing an edge filter or you execute a region processing (e.g., morphological operators) after a segmentation. To facilitate a reasonable processing all the possible operators are offered in this menu item.

2.3.9.3 Suggestions ▷ Alternatives

Since HALCON includes a large library, this menu item suggests alternative operators. Thus, you may, for example, replace mean_image with operators such as gauss_image, sigma_image, or smooth_image.

2.3.9.4 Suggestions ▷ See also

Contrary to Suggestions ▷ Alternatives, operators are offered here which have some *connection* to the current operator. Thus, the median filter (median_image) is not a direct alternative to the mean filter (mean_image). Similarly, the regiongrowing operator (regiongrowing) is no alternative for a thresholding. In any case, they offer another approach to solve a task. References might consist of pure informative nature, too: the operator gen_lowpass, which is used to create a lowpass filter in the frequency domain, is a reasonable reference to a Gaussian filter.

2.3.9.5 Suggestions ▷ Keywords

This menu item gives access to HALCON operators by using keywords which are associated with each operator. You get a window, divided into two parts, which contains all keywords on the left hand side and the selected operators on the right (see figure 2.34).

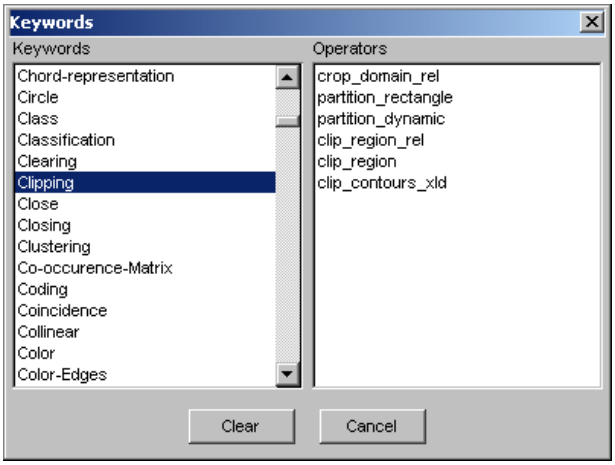

Figure 2.34: Operator suggestions according to keyword "Clipping".

After the suggestions for an operator have been generated, all keywords belonging to this operator are marked (reversed) on the left hand side of the window[4]. On the right side you will find all operators associated with at least one of these keywords. Clicking a keyword on the left list causes the addition of operators belonging to this keyword. If you want to transfer one of these operators to the operator dialog area, you click one of them with the left mouse button. Afterwards the selection window is closed.

[4]Because there are many entries in the left keyword list, you may see all marked keywords only by scrolling it.

2.3.10 Menu `Window` (Windows NT/2000/XP only)

On a Windows NT/2000/XP system, this menu offers support to manage the sub-windows of the main window, i.e., the program, operator, variable, graphics window(s), and possibly other dialogs. You see the provided menu items in figure 2.35.

Note that this menu is not supported in a UNIX environment, because here the main window has a reduced functionality.

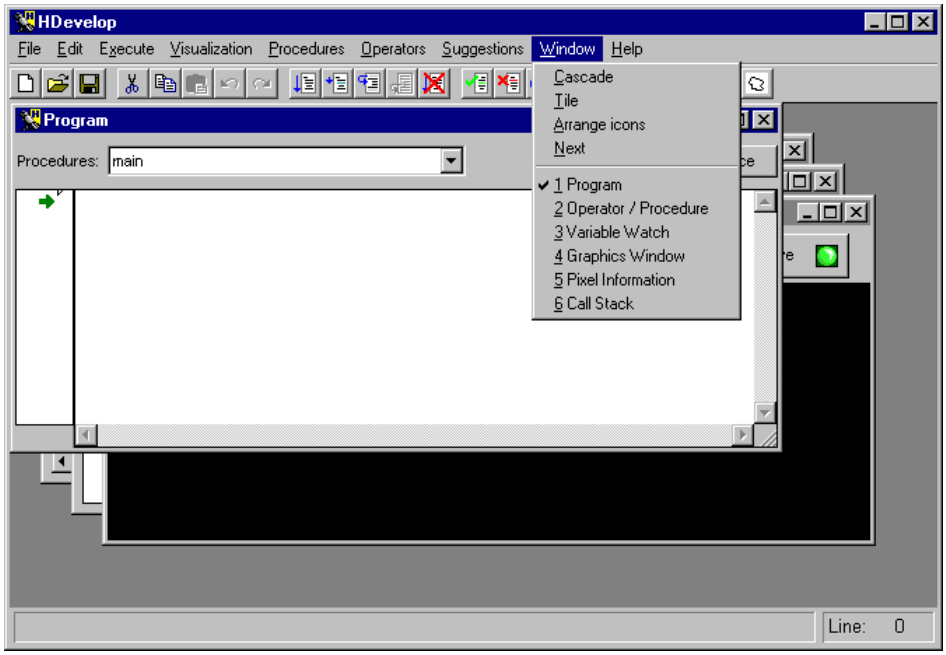

Figure 2.35: Window management functions.

2.3.10.1 `Window` ▷ `Cascade`

By selecting this item, HDevelop arranges the program, operator, variable, and graphics window in a cascade as you can see in figure 2.35.

2.3.10.2 `Window` ▷ `Tile`

When selecting this item, you see the program, operator, variable, and graphics window inside the main window. They have the same size and fit exactly in the main window. Thus, you get a global view of the windows' contents at once. Notice that the four windows may shrink depending on their size to fit in the main window. Figure 2.36 shows you the effect using this item.

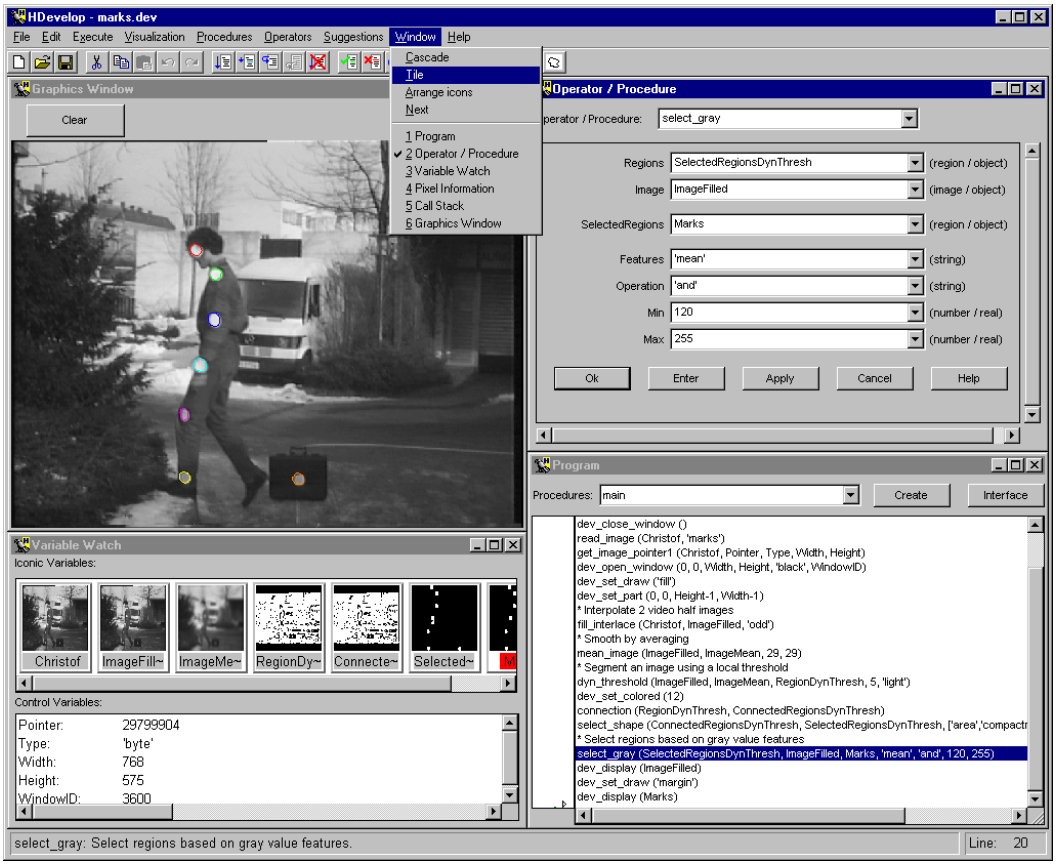

Figure 2.36: The window management function `Tile`.

2.3.10.3 `Window` ▷ `Arrange icons`

As in every system using windows, you are able to iconify and deiconify your windows. You may even move your icons on the display. This might create a confusing working environment if you are handling several graphics tools simultaneously. To get the HDevelop icons back on top of the main window's status bar, you just have to press this button.

2.3.10.4 `Window` ▷ `Next`

By using this item you move the top window in the background. Hence, it looses its window focus. Instead, the window that was only hidden by the former top window becomes the top window and gets the window focus.

2.3.10.5 `Window` ▷ `others`

If you select one of the other entries, the specified window will become the top window and gets the window focus.

2.3.11 Menu `Help`

Here you may query information about HALCON itself and all HALCON and HDevelop operators.

2.3.11.1 `Help` ▷ `About`

This menu item delivers information about the current HALCON version (see figure 2.37). Furthermore, it lists host IDs detected by the license manager (see the Installation Guide, section 3.1 on page 17, for more information).

Figure 2.37: Information about the current HALCON version.

2.3.11.2 `Help` ▷ `HALCON Operators`

This online help is based on an HTML browser (see section 6.3 on page 144). The browser will display the main page of all HALCON and HDevelop operators. It is quite easy for you to browse through this operator menu and to find the desired operator.

Another possibility of requesting information about the current operator is pressing the button `Help` inside the operator window (see page 71).

2.3.11.3 `Help` ▷ `HDevelop Language`

This menu item starts the HTML browser with a description of the language, similar to chapter 3 on page 81 of this manual.

2.3.11.4 Help ▷ HALCON News (WWW)

This menu item lets you check for the latest news about HALCON on MVTec's WWW server, e.g., whether new extension packages, frame grabber interfaces, or HALCON versions are available.

2.3.12 Tool Bar

You use most icons in this tool bar to accelerate accessing important HDevelop features. These are features which you are performing many times while working with HDevelop. Hence, there are buttons to handle your HDevelop programs and to edit them. The most important buttons are used to start and to stop a program (or parts of a program). These icons are explained in figure 2.38

	These icons are shortcuts for the following menu items: File ▷ New, File ▷ Load, File ▷ Save. For a detailed description see section 2.3.3 on page 15.
	These icons are shortcuts for the following menu items: Edit ▷ Cut, Edit ▷ Copy, Edit ▷ Paste, Edit ▷ Undo, Edit ▷ Redo. For a detailed description see section 2.3.4 on page 23.
	These icons are shortcuts for the following menu items: Execute ▷ Run, Execute ▷ Step, Execute ▷ Step Into, Execute ▷ Step Out, Execute ▷ Stop. For a detailed description see section 2.3.5 on page 26.
	These icons are shortcuts for the following menu items: Execute ▷ Activate, Execute ▷ Deactivate, Execute ▷ Reset Program. For a detailed description see section 2.3.5 on page 26.
	This icon is a shortcut for the menu item Visualization ▷ Set Parameters. For a detailed description see section 2.3.6 on page 30.
	These icons are shortcuts for the following menu items: Visualization ▷ Pixel Info, Visualization ▷ Zooming, Visualization ▷ Gray Histogram Info, Visualization ▷ Feature Histogram Info, Visualization ▷ Region Info. For a detailed description see section 2.3.6 on page 30.

Figure 2.38: The HDevelop tool bar.

2.3.13 Window Area (Windows NT/2000/XP)

On Windows NT/2000/XP systems, the window area of the main window contains all necessary windows to show your HDevelop programs, to visualize your iconic and control results, and to specify any operator's parameters. Additionally, you may open as many graphics windows as you want to get a detailed view of your iconic results.

You are free to move the windows according to your needs and preferences inside this area. You may iconify and/or deiconify them. To handle these windows in a comfortable way, HDevelop supports you with some window management functions (see section 2.3.10 on page 58).

2.3.14 Status Bar

The status bar at the bottom of the program window shows you information that is important while working with HALCON, e.g., context sensitive information about a specific user action or the operator or procedure call runtime (if time measurement has not been deactivated via the menu item `File` ▷ `Options` (see section 2.3.3.11 on page 20)).

2.4 Program Window

The program window is divided into three areas:

- At the top, you find elements for selecting or creating procedures.
- Below this, the column at the left side contains the program counter, the insertion cursor, and optionally, one or more break points.
- The main part of the program window contains the program code of the current HDevelop procedure.

These three parts are described in the following sections, but in the reverse sequence.

2.4.1 The Program Area

The main part of the program window contains the program code of the current HDevelop procedure. Here, the user has the possibility to obtain information about the inserted operators or procedure calls. A program is built up such that every line contains exactly *one* operator or procedure call with its parameters, or an assignment. An exception are the conditional constructs `if` and `ifelse`, respectively, and the loop constructs `while` and `for`. They contain two, and in case of `ifelse` three, program lines, which enclose the body. Every line starts with an operator or procedure name, which is indented, if necessary, to highlight the structure created by the above mentioned control structures. After the operator or procedure name the parameters are displayed in parentheses. Parameters are separated by commas.

The program window is used to *visualize* program lines, but not to modify them. You cannot change a program body by modifying the text directly. Editing the program text in HDevelop is done in the *operator window* (see section 2.5 on page 71). The main reason for this principle is the advantage that it facilitates providing sophisticated help. Thus, many input errors can be avoided.

To edit a line of a program, you double-click it with the left mouse button. In case of conditions and loops it is unimportant which lines (e.g., `for` or `endfor`) are selected. In any case, the head with its parameters is selected. You may edit only *one* operator or procedure call at a time.

Besides editing the parameters of a single operator or procedure call, single and multiple lines can be deleted, cut, or pasted in one step using simple mouse functions. To use this feature, one has to select one or more lines using the mouse:

- The selection of *one* line is done by clicking on it. Previously activated lines will then become deactivated.
- To activate more than one line you have to press the `<Ctrl>` key while clicking on the line. If the line is already activated it will become deactivated, while the state of all other lines remains unchanged.
- The `<Shift>` key is used to activate a sequence of lines using one mouse click: All lines between the most recent activation and the new one will become activated.

After the selection of lines, the edit function can be activated by either using the menu `Edit` (see section 2.3.4 on page 23), or the tool bar (see section 2.3.12 on page 61), or via the context menu of the column to the left (see below). Further information on the use of the mouse can be found in section 2.1 on page 11.

2.4.2 Program Counter, Insertion Cursor, and Break Points

The column to the left of the displayed program body contains the *program counter* (PC), represented as a green arrow pointing to a program line, the insertion cursor (a triangle between two program lines) and optionally one or more *break points* (BP — a red STOP sign).

The program counter resides in the line of the next operator or procedure call to execute. The insertion cursor indicates the position to insert a new program line. A break point shows the program line on which the program is stopped. In figure 2.39 you see a program and the column with the PC (indicated as an arrow), a BP, and the insertion cursor.

You may position or activate these three labels as follows:

- The PC is set by pressing the left mouse button only.
- The insertion cursor is set by pressing the left mouse button and the `<Shift>` key.
- A BP is set by pressing the left mouse button and the `<Ctrl>` key. Clicking on a break point again while pressing the `<Ctrl>` key deletes it.

Furthermore, in the column procedure calls are marked by green bars (if you did not disable the marking in the menu item `File` ▷ `Options` as described in section 2.3.3.11 on page 20). See section 2.4.3 on page 65 for an example.

By clicking into the column on the left with the right mouse button you can open a context menu, which contains shortcuts to some of the actions of the menus `Edit`, e.g., copy and paste lines, and `Execute`, e.g., activate and deactivate lines or set and clear break points. Please note that these actions behave slightly differently than their counterparts in the main menus: When called via the main menus, the actions are performed only on the selected part of the program; if nothing is selected, no action is performed. In

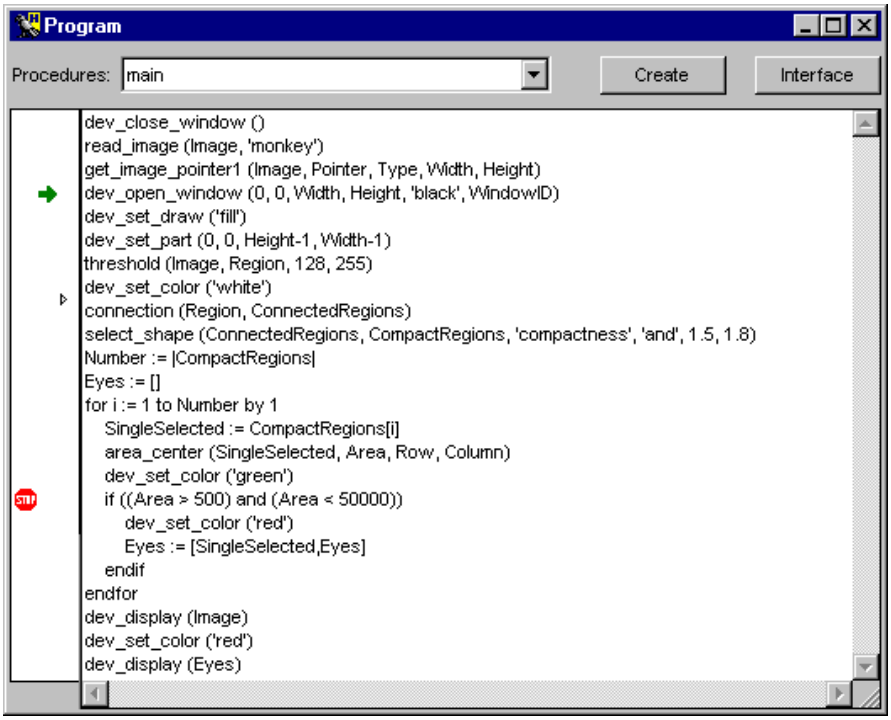

Figure 2.39: Program example with the PC (the arrow pointing to the right), insertion cursor and a break point (BP).

contrast, when an action is called via the context menu and no line is selected in the program, the action is performed for the line onto which you clicked with the right mouse button.

Note that any actions that modify the position of the PC will cause the call stack to pop all procedure calls until the current procedure call remains on top. This is relevant in case the current procedure call is not the top-most procedure call and is necessary to secure the consistency of the call stack. Modification of the PC can happen as well directly as described above or indirectly by e.g. inserting a program line above the PC in the current procedure body.

2.4.3 Creating and Editing Procedures

HDevelop always displays one procedure, the current procedure, at a time. The procedure selection box on top of the program window displays the name of the current procedure and enables it to be changed directly by the user. It contains a list of all HDevelop procedures in the current program, with the main procedure always being the first element in the list, while the other procedures are sorted alphabetically. After being selected from the list, a procedure becomes the current procedure and the corresponding procedure call becomes the current procedure call. If the selected procedure has multiple calls on the stack, the last of the procedure calls is displayed.

While the body of the current procedure is visualized in the program window, the procedure *interface* can optionally be viewed or modified in the procedure interface dialog displayed in the operator window. The buttons `Create` and `Interface` at the top of the program window perform the same action as the menu items `Procedures` ▷ `Create` and `Procedures` ▷ `Interface`, respectively. Activating one of these buttons invokes the procedure interface dialog.

2.4.3.1 Procedure Interface Dialog

The procedure interface dialog enables you to view and edit the interface of HDevelop procedures. If it is opened in order to create a new procedure (see figure 2.41), it contains a text field for the procedure name to be entered. If you edit an existing procedure (see figure 2.40), the name of the procedure is displayed in the procedure combo box on top of the dialog. A further text field can be used to enter an optional short description for the procedure. The short description of a procedure is treated like the short

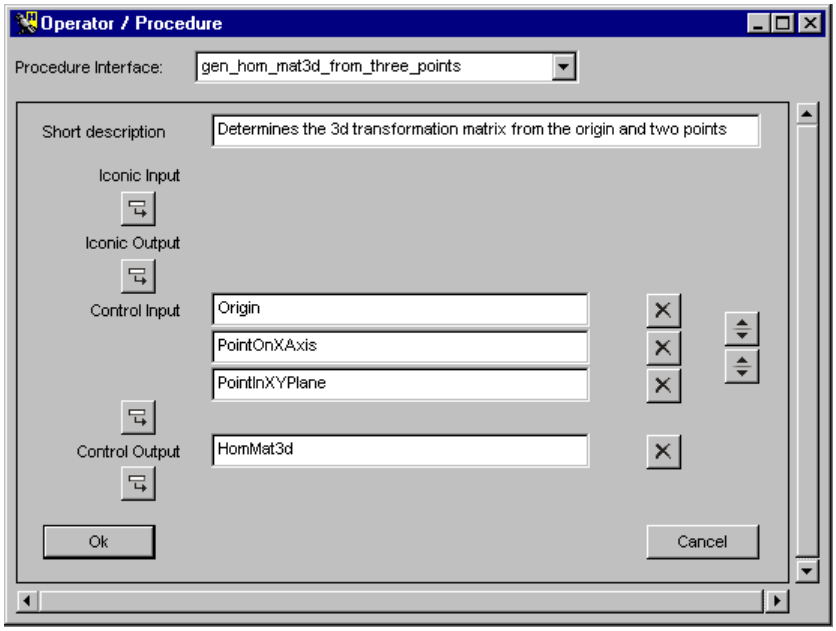

Figure 2.40: Procedure interface of an existing procedure.

information of a HALCON operator, i.e., it is displayed in HDevelop's status bar when double clicking on a procedure call in the program window or selecting a procedure from the menu.

The next part of the dialog is used for the procedure interface parameters. As mentioned earlier, HDevelop procedure interfaces have the same structure as HALCON operator interfaces, that is, they may contain parameters of the four categories iconic input, iconic output, control input, and control output in this order. The procedure interface dialog contains four separate areas that offer the necessary functionality for manipulating parameters. These areas correspond to above parameter classes and are independent of each other. Every area is marked with a label that describes the parameter class. It contains a button for inserting new parameters, which are always appended at the end of the parameter list. The latter is displayed by an array of text fields containing the parameters' names. At the right of every parameter field is a button with which the corresponding parameter can be deleted from the list. If there exist two or more parameters in a particular parameter category, the dialog contains exchange buttons between every two neighboring parameters, with which their positions in the interface can be swapped.

Activating the button `Ok` on bottom of the dialog either creates a new procedure or commits the changes made in the procedure interface, depending on whether the interface dialog was invoked in order to create a new procedure or to modify the interface of an existing procedure. In the latter case not only the interface itself might be changed but also the procedure's program body and variable lists, as new variables might have been added or existing variables might have been removed or renamed. Additionally, all calls to the procedure in the current program are checked for consistency and updated if necessary. Note that if new parameters are added to an existing procedure interface, the corresponding procedure calls are modified by adding new variables as input parameters which most likely will not be initialized at the time of the procedure call.

`Cancel` dismisses the dialog.

2.4.3.2 Creating Procedures

Depending on the corresponding procedure options and a possible selection of lines in the program window, there are different ways on how a procedure can be created. When activating the button `Create` the program lines marked in the program window are copied and inserted in the program body of the new procedure. If the last selected program line is not a return operator, a return call is added at the end of the procedure body. If no lines are selected in the program window, the newly created procedure body contains the return operator.

When creating a new procedure from selected program lines, HDevelop automatically determines suitable interface parameters for the procedure from the usage of the variables in the selected code. The following different options (set in `File` ▷ `Options`) can be used to determine the procedure parameters:

- `Only In Only Out`: Variables that are exclusively input and output variables become input and output procedure parameters, respectively.
- `Only In All Out`: Variables that are exclusively input variables become input parameters, while all output variables become output parameters. This is the default option in HDevelop.
- `All In Only Out`: This setting is symmetrical to the previous setting.
- `All In All Out`: All input and output variables become input and output procedure parameters, respectively.

The classification of variables in the selected program lines is performed separately for iconic and control variables. If a variable is an input as well as an output variable, it is assigned to the first category, i.e., the corresponding procedure parameter becomes an input parameter (see figure 2.41).

After the procedure is created and added to the program there exist two ways on how to proceed: If the option `Replace Selection By Call` (see menu item `File` ▷ `Options` (see section 2.3.3.11 on page 20)) is not activated, the newly created procedure becomes the current procedure. Otherwise, the operator dialog is opened and a suggested call to the new procedure is displayed (see figure 2.42). The variables that were used to determine the procedure interface parameters are now being offered as input parameters for the procedure call. Selecting `Ok` or `Enter` in the operator dialog replaces the selected program lines in the current procedure with a call to the newly created procedure (see figure 2.43). Selecting `Cancel` dismisses the operator dialog and the current procedure remains unchanged.

Figures 2.41, 2.42, and 2.43 show an example of how a new procedure is created from a selection of lines in the program window. In the example the options `Replace Selection By Call` and `Parameters:Only In All Out` are activated (see menu item `File` ▷ `Options` (see section 2.3.3.11 on page 20)). Figure 2.41 shows HDevelop after pressing the button `Create` in the program window. In the procedure interface dialog the name of the new procedure and the optional short description have already been entered and the name of the output object parameter has been changed from 'Christof' to 'Image'. Figure 2.42 shows the next step after activating the button `Ok` in the procedure interface dialog. By now, the new procedure has been created and added to the program and a call to the procedure is now displayed in the operator window. Note that all GUI functionality except the parameter fields and the buttons `Ok`, `Enter`, and `Cancel` in the operator dialog is disabled. Finally, in figure 2.43 the selected program lines in the current procedure have been replaced by a call to the new procedure after pressing `Enter` in the operator dialog. The contents of the operator window has been cleared and the HDevelop GUI returned to its default state.

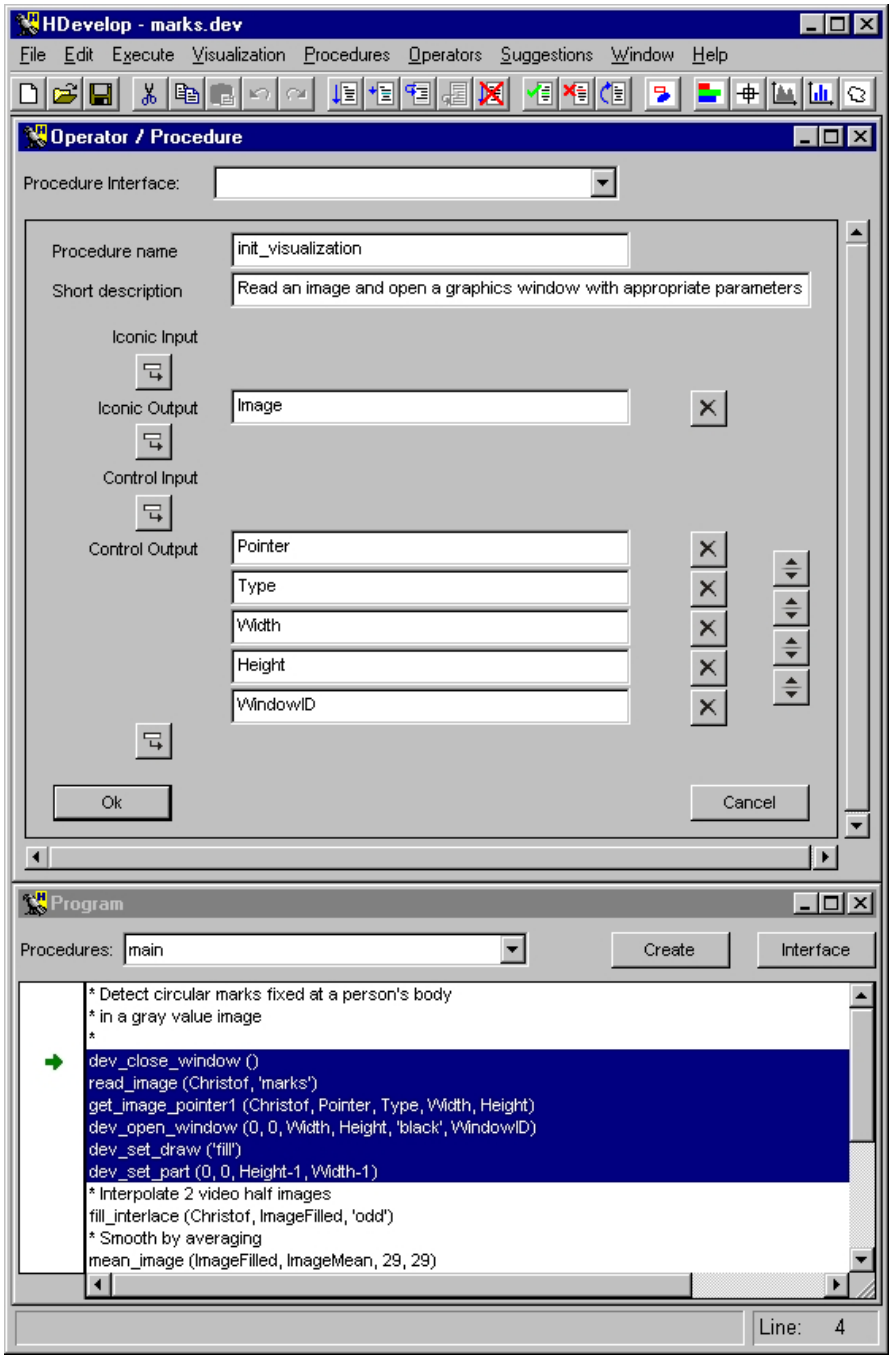

Figure 2.41: Creation of the procedure `init_visualization` from a selection of program lines.

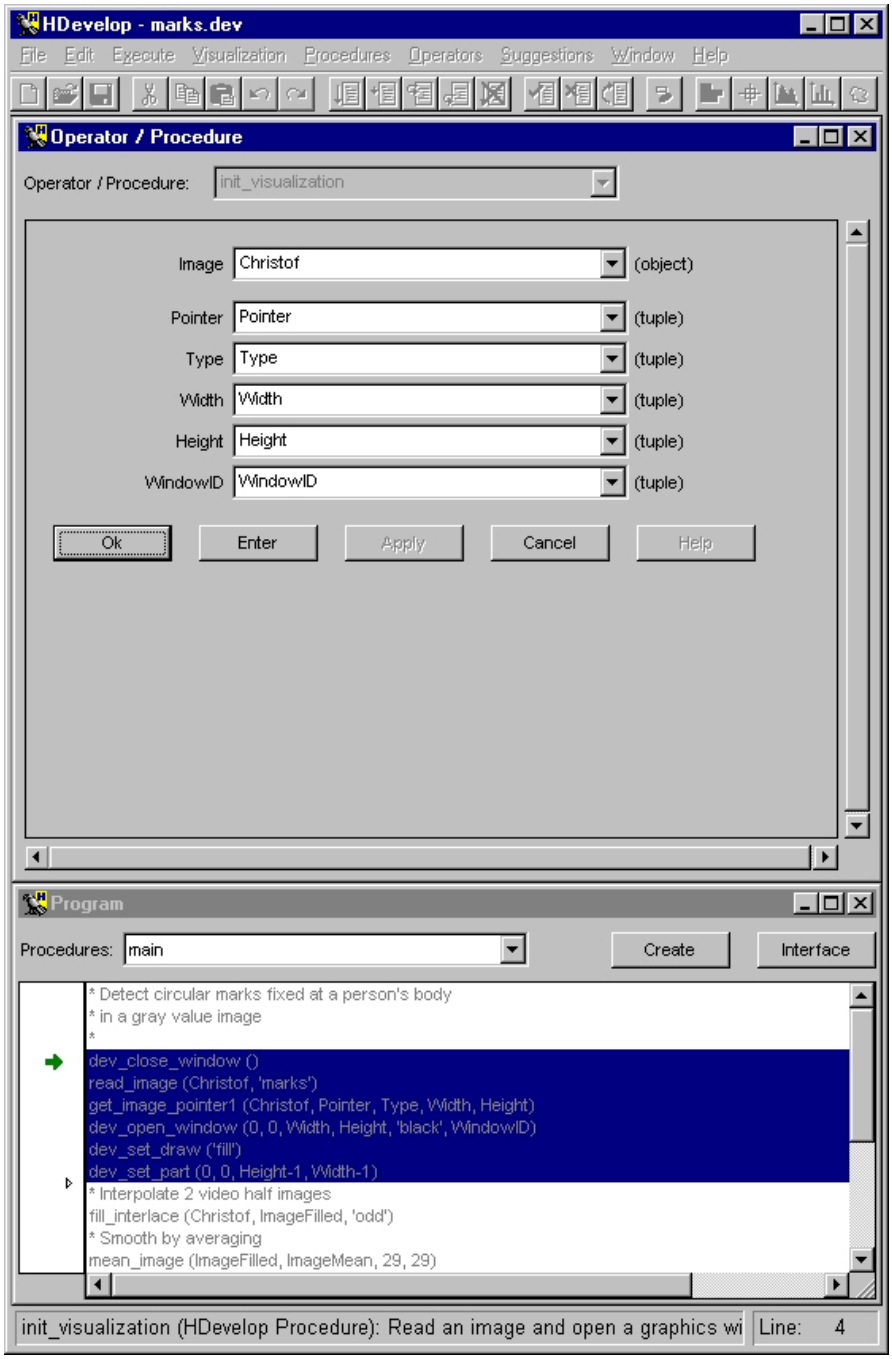

Figure 2.42: Operator window with a suggested call to the newly created procedure.

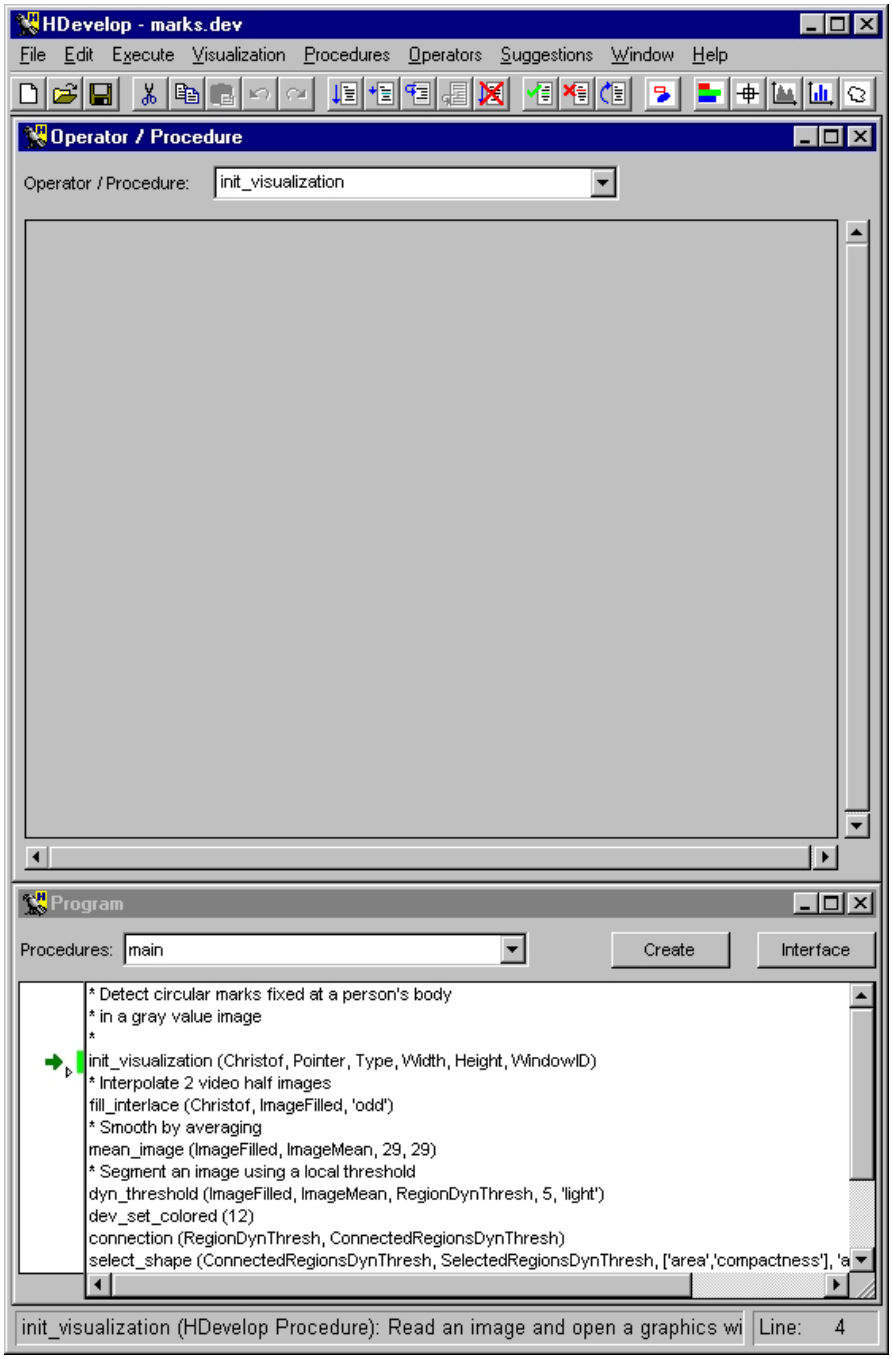

Figure 2.43: Selected program lines are replaced by a call to the newly created procedure `init_visualization`.

2.5 Operator Window

This window is mainly used to edit and display an operator or procedure call with all its parameters. Here you will obtain information about the number of the parameters of the operator or procedure, the parameter types, and parameter values. You are able to modify the parameter values according to your image processing tasks. For this you may use the values proposed by HDevelop or specify your own values.

The operator window consists of the following three parts:

- At the top you find the operator name field, with which you can select operators or procedures.
- The large area below the operator name field is called the parameter display; it is used to edit the parameters of an operator or procedure.
- The row of buttons at the bottom allows to control the parameter display.

Note that the operator window is also used to edit the interface of a procedure. This is described in section 2.4.3 on page 65.

2.5.1 Operator Name Field

The operator name field allows to select operators or procedures by specifying a substring of its name. After pressing `<Return>` or pressing the button of the combo box, the system is looking for all operators or procedures that contain the user-specified substring.

If there is an unambiguous search result, the operator or procedure is displayed immediately in the operator window. If there are several matching results, a combo box opens and displays all operators or procedures containing the specified substring (see figure 2.44). By clicking the left mouse button you select one operator and the combo box disappears. Now, the operator's parameters are shown in the operator window.

If you are already more familiar with HDevelop, it is useful to select an operator or procedure in the operator name field. However, in order to do so, you obviously have to be familiar with the operator names.

2.5.2 Parameter Display

The parameter display is the main part of the operator window. It is empty in its initial state. If you have selected an operator or procedure call, HDevelop displays its parameter data, i.e., name, number, type, and default values, in the display.

- In the first column of the operator window you find the parameter names.
- The second column consists of the text fields, which contain variable names in case of iconic and control output parameters and expressions in case of control input parameters. If you want to change the suggestions offered by the system (variable names or default values) you may do so either manually or by pressing the arrow button connected with the respective text field. This opens a list containing a selection of already defined variables and other reasonable values from the operator knowledge base. By clicking the appropriate item you set the text field and the list disappears.

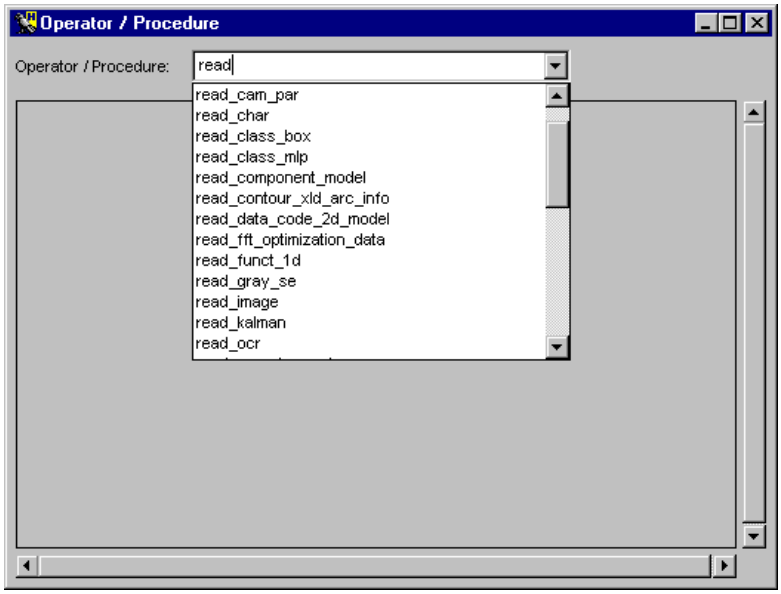

Figure 2.44: Operator selection in the operator name field.

- The third column indicates the parameter's default type in parentheses.

Please refer to the following rules on how parameters obtain their values and how you may specify them:

Iconic input parameters:
Possible inputs for these parameters are iconic variables of the corresponding list. If there is no need to execute the operator or procedure call immediately, you may even specify new variable names, i.e., names, that do not already exist in the variable window, but will be instantiated later by adding further operators or procedure calls to the program body. In any case, you have to specify iconic parameters *exclusively with variable names*. It is not possible to use expressions.

Iconic output parameters:
These parameters contain default variables, which have the same names as the parameters themselves. If a variable with the same name as the output parameter is already being used, a number is added to the name to make it unique. Because the parameter names characterize the computed result very well, you may adopt these default names in many cases. Besides this, you are free to choose arbitrary names either by yourself or by opening the list (see above). If you use a variable that already has a value, this value is deleted during execution before overwriting it with new results. It is possible to specify a variable both in an input and output position.

Control input parameters:
These parameters normally possess a default value. As an alternative, you may use the text field's button to open a combo box and to select a value suggestion. In addition, this combo box contains a list of variables that contain values of the required type. A restriction of proposed variables is especially used for parameters that contain data like file, frame grabber, or OCR handles.

Input control parameters may contain constants, variables, and expressions. Common types are integer numbers (`integer`), floating-point numbers (`real`), boolean values (`true` and `false`), and character strings (`string`).

You can also specify multiple values of these types at once by using *tuples*. This is an array of values, separated by commas and enclosed in brackets. Furthermore, you may build up expressions with these values. The possibilities of using tuples are very extensive. You may use expressions in HDevelop similar to the use of expressions in C or in Pascal. You will find a more detailed description in section 3.5 on page 85.

Control output parameters:
These parameters are handled in the same way as output object parameters. Their defaults are named as their parameter names. Other possibilities to obtain a control output variable name are either using the combo box or specifying variable names manually. You cannot use any expressions for these parameters, either.

After discussing what can be input for different parameters, it is explained *how* this is done. Nevertheless, you have to keep in mind that you need to modify a parameter only if it contains no values or if you are not satisfied with the HALCON default values.

Text input:
To specify a parameter using your keyboard is the simplest but not the most often used method. Here you have to click into a text field with the left mouse button. This activates the field and prepares it for user input. Simultaneously, the writing position is marked by a vertical bar. Now you may input numbers, strings, expressions, or variables. There are some editing functions to help you doing input: `<Backspace>` deletes the character to the left and `<Delete>` deletes the one to the right. You may also select (invert) a sequence of characters in the text field using the mouse. If there is a succeeding input, the marked region is going to be deleted first and afterwards the characters are going to be written in the text field. You can find additional editing functions in section 6.1 on page 143.

Combo box selection:
Using this input method, you can obtain rapid settings of variables and constants. To do so, you have to click the button on the text field's right side. A combo box is opened, in which you may select an item. Thus, you are able to choose a certain variable or value without risking erroneous typing. This item is transferred to the operator name field. Previous entries are deleted. Afterwards, the combo box is closed. If there are no variables or appropriate values, the combo box remains closed.

Drag and drop from variable window
You can also select a variable from the variable window as follows: Click onto the variable with the right mouse button. A context menu opens which lists the parameters for which the variable can be used as a value. Input parameters are marked by a '<', output parameter by a '>'. If you select a menu item, the variable is automatically entered in the corresponding parameter field.

2.5.3 Control Buttons

Below the parameter edit fields, you find four buttons that comprise the following functions (see figure 2.45):

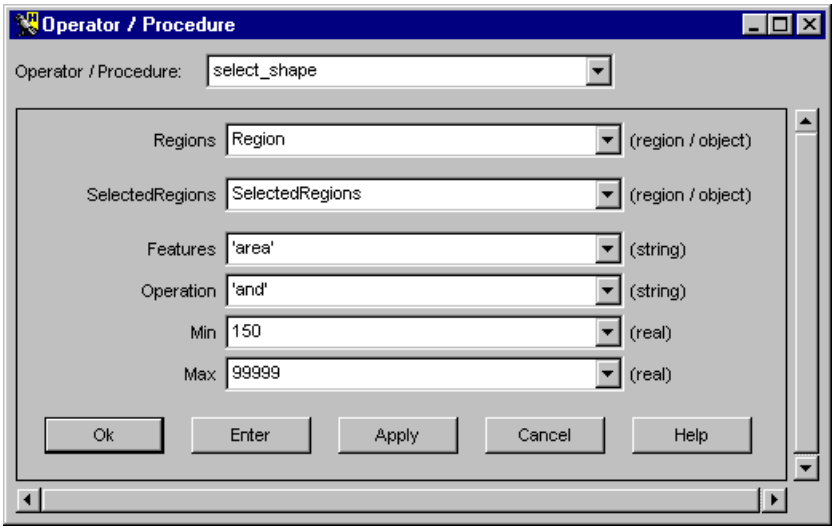

Figure 2.45: Operator window with operator `select_shape`.

`Ok`

By clicking `Ok` you execute the operator or procedure call with the specified parameters. When doing so, the execution mode depends on the position of the *program counter* (PC): If the PC is placed above the insertion position, the system executes the program from the PC until the insertion position first. *Then*, the operator or procedure call that has been edited in the operator window is executed. The reason for this is that the parameter values that are used as input values for the currently edited operator or procedure call have to be calculated. If the PC is placed at or after the insertion position, only the currently edited operator or procedure call is executed.

The operator or procedure call is entered into the program window before it is executed. After the execution, the PC is positioned on the next executable program line after the edited operator or procedure call.

The computed output parameter values are displayed in the variable window. Iconic variables are shown in the current graphics window if you haven't suppressed this option (compare section 2.3.3.11 on page 20). Afterwards, the operator window is cleared. If you did not specify all parameters or if you used wrong values, an error dialog is raised and execution is canceled. In this case, the operator window remains open to allow appropriate changes.

`Enter`

By clicking the button `Enter`, the currently edited operator or procedure call is transferred into the program window without being executed.

`Apply`
: If you click `Apply` the operator is executed with the specified parameters, but not entered into or changed in the program. This enables you to determine the optimum parameters rapidly since the operator dialog remains open, and hence you can change parameters quickly. Note that this functionality is not available for procedure calls, and thus the button is grayed out in this case.

  Unlike the button `Ok`, only the single line you edit or enter is executed, no matter where the PC is located. Thus, you have to ensure that all the input variables contain meaningful values. By pressing `Apply`, the corresponding output variables are changed or created, if necessary, to allow you to inspect their values. If you decide not to enter the line into the program body, some unused variables may thus be created. You can easily remove them by selecting `File` ▷ `Cleanup`.

`Cancel`
: Clicking `Cancel` clears the contents of the operator window. Thus, there are neither changes in the program nor in any variables.

`Help`
: Clicking `Help` invokes an appropriate help text for the selected operator. For this the system activates an HTML browser (see section 6.3 on page 144). Note that this functionality is not available for procedure calls, and thus the button is grayed out in this case.

2.6 Variable Window

There are two kinds of variables in HALCON, corresponding to the two parameter types of HALCON: iconic objects (images, regions, and XLDs) and control data (numbers, strings). The corresponding variables are called iconic and control variables. These variables may possess a value or may be undefined. An undefined variable is created, for example, when loading a program or after inserting an operator with a new variable that is not executed immediately into a program. You may access these undefined variables only by writing them. If you try to read such a variable, a runtime error occurs. If a variable obtains a value, the variable type is specified more precisely. A control variable that contains, for example, an integer is of type `integer`. This type might change to `real` or a tuple of `integer` after specifying new values for this variable. But it always remains a control variable. Similarly, this is the case for iconic variables, which may contain regions, images, or XLDs. You may assign new values to an iconic variable as often as you want to, but you cannot change its type so that it becomes a control variable.

In addition to classifying HDevelop variables by whether they are iconic or control variables, they can also be distinguished by whether they are interface parameters of the current procedure or local variables. Generally, both kinds of variables are treated equally, except that interface parameters in the variable window are marked by the prefixes '<' and '>' for input and output parameters, respectively.

New variables are created in the operator dialog area during specification of operator or procedure call parameters. Here, every sequence of characters without single quotation marks is interpreted as a variable name. If this name did not exist before, the variable is created in the operator dialog area by pressing `Ok` or `Enter`. The variable type is specified through the type of the parameter where it was used for the first time: Variables that correspond to an iconic object parameter create an iconic variable; variables for a control parameter create a control variable. Every time an operator or procedure call is executed, the results are stored in variables connected to its output parameters. This is achieved by first deleting the contents of the variable and then assigning the new value to it.

The variable window is a kind of watch window used in window-oriented debuggers. Inside this window you are able to keep track of variable values. Corresponding to the two variable types, there are two areas in the variable window. One for iconic data (above) and the other for control data (below) (see figure 2.46).

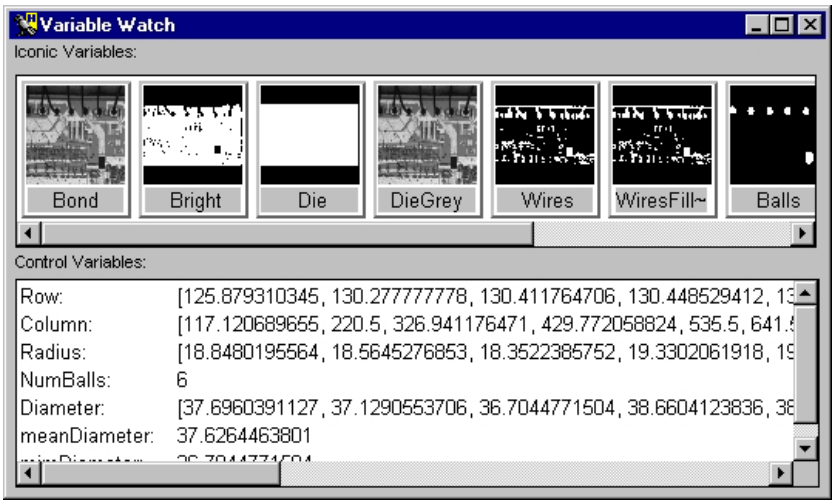

Figure 2.46: Variable window for both iconic and control data.

All computed variables are displayed showing their iconic or control values (if the automatic update has not been switched off, see section 2.3.3.11 on page 20). The sorting order of the variable names can be set as described in section 2.3.3.11 on page 20. In case of a tuple result which is too long, the tuple presentation is shortened, indicated by three dots.

2.6.1 Area for Iconic Data

Here you can see iconic variables. They are represented by icons, which contain an image, a region or an XLD, depending on the current value. The icons are created depending on the type of data according the following rules:

- For images the icon contains a zoomed version of them, filling the icon completely. Due to the zooming onto the square shape of the icon, the aspect ratio of the small image might be wrong. If there is more than one image in the variable, only the *first* image is used for the icon. Similarly, for multi-channel images only the *first* channel is used. The domain of the image is ignored.
- Regions are displayed by first calculating the smallest surrounding rectangle and then zooming it so that it fills the icon using a border of one pixel. In contrast to images, the aspect ratio is always correct. This can lead to black bars at the borders. The color used to draw the region is always white without further modifications (except zooming).
- XLD data is displayed using the coordinate system of the largest image used so far. The color used for XLD objects is white on black background.

Because of the different ways of displaying objects, you have to be aware that the coordinates cannot be compared. The variable name is positioned below each icon. They are displayed in the variable window in the order of creation from left to right. If there is not enough space, a horizontal scrollbar is created, which you can use to scroll the icons.

Clicking on an icon with the mouse will select this icon. This is indicated by the black background for the icon name. For an activated icon all operators or procedure calls that use the corresponding variable are marked in the program area with a black rectangle on the left.

Double-clicking with the left mouse button on an icon displays the data in the active graphics window. If you use images of different sizes in a program, the system uses the following output strategy for an automatic adaption of the zooming: Every window keeps track of the size of the most recently displayed image. If you display an image with a different size, the system modifies the graphics window coordinate system in a way that the image is visible completely in the graphics window. If a partial zooming has been activated before (see section 2.7), it is going to be suppressed.

Normally, regions, images, and XLDs are represented in variable icons. Besides this there are three exceptions, which are shown by special icons:

- Undefined variables are displayed as a question mark (?) icon. You may *to write* but not read them, because they do not have any values.
- Brackets ([]) are used if a variable is instantiated but does not contain an iconic object (empty tuple). This may be the case using operators like `select_shape` with "wrong" specified thresholds or using operator `gen_empty_obj`. Such a value might be reasonable if you want to collect iconic objects in a variable gradually in a loop using `concat_obj`. Here, an empty tuple is used as starting value for the loop.
- A last exception is an *empty region*. This is *one* region that does not contain any pixels (points), i.e., the area (number of points) is 0. You must not confuse this case with the empty tuple, because there the area is not defined. The empty region is symbolized by an empty set icon (∅).

2.6.2 Area for Control Data

To the right of the variable name you find their values in the default representation[5]. If you specify more than one value for one variable (tuple), they are separated by commas and enclosed by brackets. If the number of values exceeds an upper limit, the output is clipped. This is indicated by three dots at the end of the tuple. For undefined variables, their name and a `?` are shown in the variable field. An empty tuple is represented by `[]`. Both exceptions use the same symbols as the corresponding cases for the iconic variables.

Clicking on a variable will select it. Similarly to iconic variables, all program lines that use this variable are then marked with a black rectangle on the left.

Double-clicking a control variable opens a window that displays all its values. In most cases this will be a dialog containing a scrolled list. This is helpful if you have tuple variables with a large number of values that you want to inspect.

[5] You have to keep in mind that a floating point number without significant fractional part is represented as an integer (e.g., 1.0 is represented as `1`).

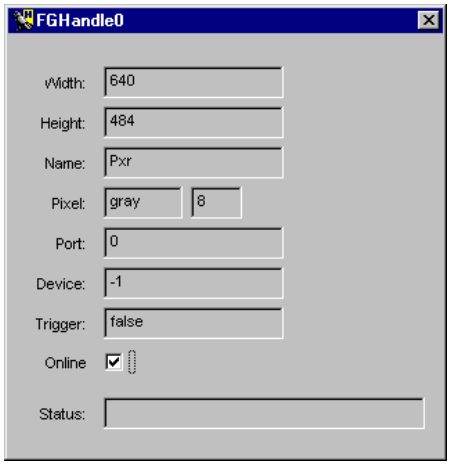

Figure 2.47: Variable inspection for frame grabber handles.

For a frame grabber handle, a dialog representing basic frame grabber parameters is opened (see figure 2.47). Here you find the size, name, device, port, and other features of the frame grabber. The toggle button `Online` allows to grab images continuously and to display them in the active graphics window. If an error occurs during grabbing, it is displayed in the status bar of the dialog. At most one of these frame grabber dialogs can be opened at the same time.

2.7 Graphics Window

This window displays iconic data. It has the following properties:

- The user may open several graphics windows.
- The active graphics window is shown by the green dot in the `Active` button.
- Pressing the `Clear` button clears the graphics window content and the history of the window.
- You close a graphics window using the close button of the window frame.

Figure 2.48 shows an example for a graphics window.

The the origin of the graphics window is the upper left corner with the coordinates (0,0). The x values (column) increase from left to right, the y values increase from top to bottom. Normally, the coordinate system of the graphics window corresponds to the the most recently displayed image, which is automatically zoomed so that every pixel of the image is visible. The coordinate system can be changed interactively using the menu `Visualization` ▷ `Set Parameters...` ▷ `Zoom` (see section 2.3.6 on page 30) or with the operator `dev_set_part` (see section 2.3.8.2 on page 51). Every time an image with another size is displayed, the coordinate system will be adapted automatically.

Each window has a history that contains all

Figure 2.48: HDevelop's graphics window.

- objects and
- display parameters

that have been displayed or changed since the most recent `Clear` or display of an image. This history is used for redrawing the contents of the window. The history is limited to a maximum number of 30 "redraw actions", where one redraw action contains all objects of one displayed variable.

Other output like text or general graphics like `disp_line` or `disp_circle` or iconic data that displayed using HALCON operators like `disp_image` or `disp_region` are *not* part of the history, and are *not* redrawn. Only the object classes image, region, and XLD that are displayed with the HDevelop operator `dev_display` or by double clicking on an icon are part of the history.

You may change the size of the graphics window interactively by "gripping" the window border with the mouse. Then you can resize the window by dragging the mouse pointer. After this size modification the window content is redisplayed. Now you see the same part of the window with changed zoom.

The menu area of the graphics window has an additional function: If the mouse cursor is in this area the look up table of the window is reactivated. This is necessary if other programs use their own look up table. Thus if there is a “strange” graphics window presentation, you may load the proper look up table by placing the mouse near the buttons.

If you want to specify display parameters for a window you may select the menu item `Visualization` in the menu bar. Here you can set the appropriate parameters by clicking the desired item (see section 2.3.6 on page 30). The parameters you have set in this way are used for *all* windows. The effects of the new parameters will be applied directly to the *last* object of the window history and alter its parameters only.

For further information on parameter effects please refer to the appropriate HALCON operators in the reference manual.

Chapter 3

Language

The following chapter introduces the syntax and the semantics of the HDevelop language. In other words, it illustrates what you can enter into a parameter slot of an operator or procedure call. In the simplest case this is the name of a variable, but it might also be an expression like `sqrt(A)`. Besides, control structures (like loops) and the semantics of parameter passing are described.

Chapter 5 on page 119 explains the application of this language in image analysis. Note that the HALCON operators themselves are not described in this chapter. For this purpose refer to the HALCON reference manual. All program examples used in this chapter can also be found in the directory `%HALCONROOT%\examples\hdevelop\Manuals\HDevelop`.

3.1 Basic Types of Parameters

HALCON distinguishes two kinds of data: control data (numerical/string) and iconic data (images, regions, etc.).

By further distinguishing *input* from *output parameters*, we get four different kinds of parameters. These four kinds always appear in the same order in the HDevelop parameter list. Table 3.1 shows their order of appearance.

iconic	input
iconic	output
control	input
control	output

Table 3.1: Order of appearance of the four basic parameter types.

As you see, iconic input objects are always passed as the first parameter(s), followed by the iconic output objects. The iconic data is followed by the control data, and again, the input parameters succeed the output parameters. Each parameter is separated from its neighbours by a comma:

```
read_image (Image, 'Name')
area_center (Region, Area, Row, Column)
mean_image (Image, Mean, 11, 11)
```

In the above example the operator read_image has one output parameter for iconic objects (Image) and one input control parameter (filename). area_center accepts regions as input (iconic) and three control parameters as output (Area, Row, Column). The filter operator mean_image has one iconic parameter as input and one as output. Its two input control parameters specify the size of the filter mask.

Input control parameters can either be variables, constants or even complex expressions. An expression is evaluated *before* it is passed to a parameter that receives the result of the evaluation. Since iconic objects always are represented by variables all iconic parameters only accept variables. Control output parameters must always contain variables, too, as they store the results of an operator evaluation.

3.2 Control Types and Constants

All non-iconic data is represented by so called *control data* (numerical/string) in HDevelop. The name is derived from their respective functions within HALCON operators where they *control* the behaviour (the effect) of image processing operators (e.g., thresholds for a segmentation operator). Control parameters in HDevelop may contain arithmetic or logical operations. A control data item can be of one of the following types: integer, real, boolean, and string.

integer and real
: The types integer and real are used under the same syntactical rules as in C. Integer numbers can be input in the standard decimal notation, in hexadecimal by prefixing the number with 0x, and in octal by prefixing the number with 0. For example:

```
4711
-123
0xfeb12
073421
73.815
0.32214
.56
-17.32e-122
32E19
```

Data items of type integer or real are converted to their machine-internal representations: real becomes the C-type double (8 bytes) and integer becomes the C-type long (4 or 8 bytes).

string
: A String (string) is a sequence of characters that is enclosed in single quotes ('). The maximum string length is limited to 1024 characters. Special characters, like the line feed, are represented in the C-like notation, as you can see in table 3.2 (see the reference of the C language for comparison).

Examples of strings are shown in table 3.3.

Meaning	Abbreviation	Notation
line feed	NL (LF)	\n
horizontal tabulator	HT	\t
vertical tabulator	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
bell	BEL	\a
backslash	\	\\
single quote	’	\’

Table 3.2: Surrogates for special characters.

String	Meaning
’Hugo’	letters
’10.9’	numbers (not real)
’Text...\n’	NL at the end of the string
’\t Text1 \t Text2’	two tabs in a text
’Sobel\’s edge-filter’	single quote within the text
’c:\\Programs\\MVTec\\Halcon\\images’	Directory

Table 3.3: String examples.

boolean

The constants true and false belong to the type boolean. The value true is internally represented by the number 1 and the value false by 0. This means, that in the expression Val := true the effective value of Val is set to 1. In general, every integer value $\neq 0$ means true. Please note that some HALCON operators take logical values for input (e.g., set_system). In this case the HALCON operators expect string constants like ’true’ or ’false’ rather than the represented values true or false.

constants

There are constants for the return value (result state) of an operator. The constants can be used together with the operator dev_error_var and dev_set_check. These constants represent the normal return value of an operator, so called *messages*. For errors no constants are available[1].

Constant	Meaning	Value
H_MSG_TRUE	No error; for tests: (true)	2
H_MSG_FALSE	For tests: false	3
H_MSG_VOID	No result could be computed	4
H_MSG_FAIL	Operator did not succeed	5

Table 3.4: Return values for operators.

[1] There exist more than 400 error numbers internally (see the Extension Package Programmer’s Manual, appendix A on page 117.

In table 3.4 all return messages can be found.

The control types are only used within the generic HDevelop type *tuple*. A tuple of length 1 is interpreted as an atomic value. A tuple may consist of several numerical data items with *different* types. The standard representation of a tuple is a listing of its elements included into brackets (see figure 3.1).

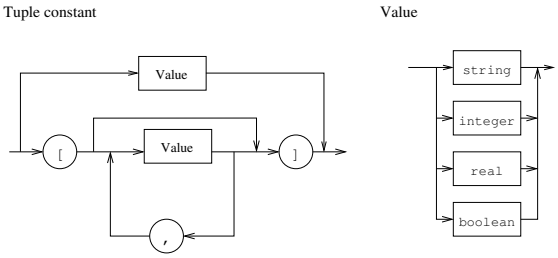

Figure 3.1: The syntax of tuple constants.

`[]` specifies the empty tuple. A tuple with just one element is to be considered as a special case, because it can either be specified in the tuple notation or as an atomic value: `[55]` defines the same constant as `55`. Examples for tuples are:

```
[]
4711
0.815
'Text'
[16]
[100.0,100.0,200.0,200.0]
['FileName','Extension']
[4711,0.815,'Hugo']
```

The maximum length of a tuple is limited to 1000000.

3.3 Variables

Names of variables are built up as usual by composing letters, digits and the underscore `'_'`. The maximum length of a variable name is limited to 256 characters. The kind of a variable (iconic or control variable) depends on its position in the parameter list in which the variable identifier is used for the first time (see also section 3.1 on page 81). The kind of the variable is determined during the input of the operator parameters: whenever a new identifier appears, a new variable with the same identifier is created. Control and iconic variables must have different names. The value of a variable (iconic or control) is undefined until the first assignment defines it (the variable hasn't been instantiated yet). A read access to an undefined variable leads to a runtime error (`Variable <x> not instantiated`).

HDevelop provides a pre-defined variable named `_` (single underscore). You can use this variable for output control parameters whose value you are not interested in. Please note that it is not allowed to use

this variable for HDevelop-specific operators (chapters `Control` and `Develop`, see the appendix for a complete list).

Instantiated variables contain tuples of values. Depending on the kind of the variable the data items are either iconic objects or control data. The length of the tuple is determined dynamically by the performed operation. A variable can get new values any number of times, but once a value has been assigned the variable will always keep being instantiated, unless you select the menu item `Execute` ▷ `Reset Program`. The content of the variable is deleted before the variable is assigned with new values.

The concept of different kinds of variables allows a first ("coarse") typification of variables (control or iconic data), whereas the actual type of the data (e.g., `real`, `integer`, `string`, etc.) is undefined until the variable gets assigned with a concrete value. Therefore, it is possible that the type of a new data item differs from that of the old.

3.4 Operations on Iconic Objects

Iconic objects are exclusively processed by HALCON operators. HALCON operators work on tuples of iconic objects, which are represented by their surrogates in the HALCON data management. The results of those operators are again tuples of iconic objects or control data elements. For a detailed description of the HALCON operators refer to the HALCON reference manual and the remarks in section 3.5.3 on page 88.

3.5 Expressions for Input Control Parameters

In HDevelop, the use of expressions is limited to control input parameters; all other kinds of parameters must be assigned by variables.

3.5.1 General Features of Tuple Operations

This chapter is intended to give you a short overview over the features of tuples and their operations. A more detailed description of each operator mentioned here is given in the following sections.

Please note that in all following tables variables and constants have been substituted by letters. These letters give information about possible limitations of the areas of definition. A single letter (inside these tables) represents a data type. Operations on these symbols can only be applied to parameters of the indicated type or to expressions that return a result of the indicated type. To begin with, table 3.5 specifies the names and types of the symbolic names.

Operators are normally described assuming atomic tuples (tuples of length 1). If the tuple contains more than one element, most operators work as follows:

- If one of the tuples is of length one, all elements of the other tuples are combined with that single value for the chosen operation.
- If both tuples have a length greater than one, both tuples must have the same length (otherwise a runtime error occurs). In this case, the selected operation is applied to all elements with the same index. The length of the resulting tuples is identical to the length of the input tuples.

Symbol	Types
i	integer
a	arithmetic, that is: integer or real
l	boolean
s	string
v	all types (atomic)
t	all types (tuple)

Table 3.5: Symbolic variables for the operation description.

- If one of the tuples is of length 0 ([]), a runtime error occurs.

In table 3.6 you can find some examples for arithmetic operations with tuples. In this example you

Input	Result
5 * 5	25
[5] * [5]	25
[1,2,3] * 2	[2,4,6]
[1,2,3] * 2.1 + 10	[12.1,14.2,16.3]
[1,2,3] * [1,2,3]	[1,4,9]
[1,2,3] * [1,2]	runtime error
'Text1' + 'Text2'	'Text1Text2'
17.23 + ' Text'	'17.23 Text'
'Text1 ' + 99 + ' Text2'	'Text1 99 Text2'
'Text ' + 3.1 * 2	'Text 6.2'
3.1 * (2 + ' Text')	runtime error
3.1 + 2 + ' Text'	'5.1 Text'
3.1 + (2 + ' Text')	'3.12 Text'
'Text ' + 2.1 + 3	'Text 2.13'

Table 3.6: Examples for arithmetic operations with tuples and strings.

should pay special attention to the order in which the string concatenations are performed.

3.5.2 Assignment

In HDevelop, an assignment is treated like an operator. To use an assignment you have to select the operator `assign(Input, Result)`. This operator has the following semantics: It evaluates `Input` (right side of assignment) and stores it in `Result` (left side of assignment). However, in the program text the assignment is represented by the usual syntax of the assignment operator: ':='. The following example outlines the difference between an assignment in C syntax and its transformed version in HDevelop:

The assignment in C syntax

```
u = sin(x) + cos(y);
```

is defined in HDevelop using the assignment operator as

```
assign(sin(x) + cos(y), u)
```

which is displayed in the program window as:

```
u := sin(x) + cos(y)
```

If the result of the expression doesn't need to be stored into a variable, the expression can directly be used as input value for any operator. Therefore, an assignment is necessary only if the value has to be used several times or if the variable has to be initialized (e.g., for a loop).

A second assignment operator is available: `insert(Input,Value,Index,Result)`. It is used to assign tuple elements. If the first input parameter and the first output parameter are identical, the call:

```
insert (Areas, Area, Radius-1, Areas)
```

is not presented in the program text as an operator call, but in the more intuitive form as:

```
Areas[Radius-1] := Area.
```

To construct a tuple with `insert`, normally an empty tuple is used as initial value and the elements are inserted in a loop:

```
Tuple := []
for i := 0 to 5 by 1
  Tuple[i] := sqrt(real(i))
endfor
```

As you can see from this example, the indices of a tuple start at 0.

An insertion into a tuple can generally be performed in one of the following ways:

1. In case of appending the value at the 'back' or at the 'front', the concatenation can be used. Here the operator `assign` is used with the following parameters:

   ```
   assign([Tuple,NewVal],Tuple)
   ```

   which is displayed as

   ```
   Tuple := [Tuple,NewVal]
   ```

2. If the index position is somewhere in between, the operator `insert` has to be used. It takes the following arguments as input: first the tuple in which the new value should be inserted; then the new value and after that the index position as the third input parameter. The result (the fourth parameter) is almost identical with the input tuple, except of the new value at the defined index position.

In the following example regions are dilated with a circle mask and afterwards the areas are stored into the tuple `Areas`. In this case the operator `insert` is used.

```
read_image (Mreut, 'mreut')
threshold (Mreut, Region, 190, 255)
Areas  := []
for  Radius := 1 to 50 by 1
  dilation_circle (Region, RegionDilation, Radius)
  area_center (RegionDilation, Area, Row, Column)
  Areas[Radius-1] := Area
endfor
```

Please note that first the variable `Areas` has to be initialized in order to avoid a runtime error. In the example `Areas` is initialized with the empty tuple (`[]`). Instead of `insert` the operator `assign` with tuple concatenation

```
Areas := [Areas,Area]
```

could be used, because the element is appended at the back of the tuple.

More examples can be found in the program `assign.dev`.

3.5.3 Basic Tuple Operations

A basic tuple operation may be selecting one or more values, combining tuples (concatenation) or reading the number of elements.

`[t,t]`	concatenation of tuples
`|t|`	number of elements
`t[i]`	selection of an element
`t[i:i]`	selection of (a part of) a tuple
`subset(t1,t2)`	selection of a subset of elements of a tuple
`find(t1,t2)`	indices of all occurrences of `t1` within `t2`
`uniq(t)`	discard all but one of successive identical elements from a tuple

Table 3.7: Basic operations on tuples.

The concatenation accepts one or more variables or constants as input. They are all listed between the brackets, separated by commas. The result again is a tuple.

`[`t_1`,`t_2`]` is the concatenation of tuple t_1 and t_2. Example:

`[[5,'Text'],[5.9]]` $\mapsto$ `[5,'Text',5.9]`

So even the following holds: `[[t]] = [t] = t`.

`|t|` returns the number of elements of a tuple. The indices of elements range from zero to the number of elements minus one (i.e., `|t|-1`). Therefore, the selection index has to be within this range.[2]

[2] Please note that the index of objects (e.g., `select_obj`) ranges from 1 to the number of elements.

```
Tuple := [V1,V2,V3,V4]
for i := 0 to |Tuple|-1 by 1
  fwrite_string (FileHandle,Tuple[i]+'\n')
endfor
```

Note that these direct operations cannot be used for iconic tuples, i.e., iconic objects cannot be selected from a tuple using `[]` and their number cannot be directly determined using `||`. For this purpose, however, HALCON operators are offered that carry out the equivalent tasks. In table 3.8 you can see tuple operations that work on control data and their counterparts that work on iconic data. In the table the symbol `t` represents a control tuple, and the symbols `p` and `q` represent iconic tuples. Further examples can be found in the program `tuple.dev`.

control	iconic
`[]`	`gen_empty_obj ()`
`|t|`	`count_obj (p, num)`
`[t1,t2]`	`concat_obj (p1, p2, q)`
`t[i]`	`select_obj(p, q, i+1, 1)`
`t[i:j]`	`copy_obj(p, q, i+1, j-i+1)`

Table 3.8: Equivalent tuple operations for control and iconic data.

3.5.4 Tuple Creation

The simplest way to create a tuple, as mentioned in section 3.2 on page 82, is the use of constants together with the operator `assign`:

```
assign([],empty_tuple)
assign(4711,one_integer)
assign([4711,0.815],two_numbers)
```

This code is displayed as

```
empty_tuple := []
one_integer := 4711
two_numbers := [4711,0.815]
```

This is useful for constant tuples with a fixed (small) length. More general tuples can be created by successive application of the concatenation or the operator `insert` together with variables, expressions or constants. If we want to generate a tuple of length 100, where each element has the value 4711, it might be done like this:

```
assign([],tuple)
for i := 1 to 100 by 1
  assign([tuple,4711],tuple)
endfor
```

which is transformed to

```
tuple := []
for i := 1 to 100 by 1
  tuple := [tuple,4711]
endfor
```

Because this is not very convenient a special function called `gen_tuple_const` is available to construct a tuple of a given length, where each element has the same value. Using this function, the program from above is reduced to:

```
assign(gen_tuple_const(100,4711),tuple)
```

which is displayed as

```
tuple := gen_tuple_const(100,4711)
```

If we want to construct a tuple with the same length as a given tuple there are two ways to get an easy solution, The first one is based on `gen_tuple_const`:

```
assign(gen_tuple_const(|tuple_old|,4711),tuple_new)
```

which is displayed as

```
tuple_new := gen_tuple_const(|tuple_old|,4711)
```

The second one is a bit tricky and uses arithmetic functions:

```
assign((tuple_old * 0) + 4711,tuple_new)
```

which is displayed as

```
tuple_new := (tuple_old * 0) + 4711
```

Here we get first a tuple of the same length with every element set to zero. Then we add the constant to each element.

In the case of tuples with different values we have to use the loop version to assign the values to each position:

```
assign([],tuple)
for i := 1 to 100 by 1
  assign([tuple,i*i],tuple)
endfor
```

which is displayed as

```
tuple := []
for i := 1 to 100 by 1
  tuple := [tuple,i*i]
endfor
```

In this example we construct a tuple with the square values from 1^2 to 100^2.

3.5.5 Simple Arithmetic Operations

Table 3.9 shows an overview of the available simple arithmetic operations.

All operations are left-associative, except the right-associative unary minus operator. The evaluation usually is done from left to right. However, parentheses can change the order of evaluation and some operators have a higher precedence than others (see section 3.5.14 on page 99).

`a / a`	division
`a * a`	multiplication
`a % a`	modulus
`v + v`	addition and *concatenation* of strings
`a - a`	subtraction
`-a`	negation

Table 3.9: Arithmetic operations.

The arithmetic operations in HDevelop match the usual definitions. Expressions can have any number of parentheses.

The division operator (`a / a`) can be applied to `integer` as well as to `real`. The result is of type `real`, if at least one of the operands is of type `real`. If both operands are of type `integer` the division is an integer division. The remaining arithmetic operators (multiplication, addition, subtraction, and negation) can be applied to either `integer` or `real` numbers. If at least one operand is of type `real`, the result will be a `real` number as well. In the following example.

```
V1 := 4/3
V2 := 4/3.0
V3 := (4/3) * 2.0
```

`V1` is set to 1, `V2` to 1.3333333, and `V3` to 2.0. Simple examples can be found in the program `arithmetic.dev`.

3.5.6 Bit Operations

This section describes the operators for bit processing of numbers. The operands have to be integers.

The result of `lsh(i1,i2)` is a bitwise left shift of `i1` that is applied `i2` times. If there is no overflow this is equivalent to a multiplication by 2^{i2}. The result of `rsh(i1,i2)` is a bitwise right shift of `i1` that

`lsh(i,i)`	left shift
`rsh(i,i)`	right shift
`i band i`	bitwise and
`i bor i`	bitwise or
`i bxor i`	bitwise xor
`bnot i`	bitwise complement

Table 3.10: Bit operations.

is applied `i2` times. For non-negative `i1` this is equivalent to a division by 2^{i2}. For negative `i1` the result depends on the used hardware. For `lsh` and `rsh` the result is undefined if the second operand has a negative value or the value is larger than 32. More examples can be found in the program `bit.dev`.

3.5.7 String Operations

There are several string operations available to modify, select and combine strings. Furthermore, some operations allow to convert numbers (`real` and `integer`) to strings.

`v$s`	string conversion
`v + v`	*concatenation* of strings and addition
`strchr(s,s)`	search character in string
`strstr(s,s)`	search substring
`strrchr(s,s)`	search character in string (reverse)
`strrstr(s,s)`	search substring (reverse)
`strlen(s)`	length of string
`s{i}`	selection of one character
`s{i:i}`	selection of substring
`split(s,s)`	splitting in substrings

Table 3.11: String operations.

$ converts numbers to strings or modifies strings. The operator has two parameters: The first one (left of the $) is the number that has to be converted. The second one (right of the $) specifies the conversion. This format string consists of the following four parts

```
<flags><field width><precision><conversion characters>
```

So a conversion might look like

```
1332.4554 $ '.6e'
```

`flags` Zero or more flags, in any order, which modify the meaning of the conversion specification. Flags may consist of the following characters:

- The result of the conversion is left justified within the field.

+ The result of a signed conversion always begins with a sign, + or -.

<space> If the first character of a signed conversion is not a sign, a space character is prefixed to the result. This means that if the space flag and + flag both appear, the space flag is ignored.

The value is to be converted to an "alternate form". For d and s conversions, this flag has no effect. For o conversion (see below), it increases the precision to force the first digit of the result to be a zero. For x or X conversion (see below), a non- zero result has 0x or 0X prefixed to it. For e, E, f, g, and G conversions, the result always contains a radix character, even if no digits follow the radix character. For g and G conversions, trailing zeros are not removed from the result, contrary to usual behavior.

field width An optional string of decimal digits to specify a minimum field width. For an output field, if the converted value has fewer characters than the field width, it is padded on the left (or right, if the left-adjustment flag, - has been given) to the field width.

precision The precision specifies the minimum number of digits to appear for the d, o, x, or X conversions (the field is padded with leading zeros), the number of digits to appear after the radix character for the e and f conversions, the maximum number of significant digits for the g conversion, or the maximum number of characters to be printed from a string in s conversion. The precision takes the form of a period . followed by a decimal digit string. A null digit string is treated as a zero.

conversion characters A conversion character indicates the type of conversion to be applied:

d,o,x,X The integer argument is printed in signed decimal (d), unsigned octal (o), or unsigned hexadecimal notation (x and X). The x conversion uses the numbers and letters 0123456789abcdef, and the X conversion uses the numbers and letters 0123456789ABCDEF. The precision component of the argument specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of 0 is no characters.

f The floating-point number argument is printed in decimal notation in the style [-]dddrddd, where the number of digits after the radix character, r, is equal to the precision specification. If the precision is omitted from the argument, six digits are output; if the precision is explicitly 0, no radix appears.

e,E The floating-point-number argument is printed in the style [-]drddde+dd, where there is one digit before the radix character, and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the precision is 0, no radix character appears. The E conversion character produces a number with E introducing the exponent instead of e. The exponent always contains at least two digits. However, if the value to be printed requires an exponent greater than two digits, additional exponent digits are printed as necessary.

g,G The floating-point-number argument is printed in style f or e (or in style E in the case of a G conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted; style e is used only if the exponent resulting from the conversion is less than -h or greater than or equal to the precision. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit.

`s` The argument is taken to be a string, and characters from the string are printed until the end of the string or the number of characters indicated by the precision specification of the argument is reached. If the precision is omitted from the argument, it is interpreted as infinite and all characters up to the end of the string are printed.

`b` Similar to the s conversion specifier, except that the string can contain backslash-escape sequences which are then converted to the characters they represent.

In no case does a nonexistent or insufficient field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.

Examples for the string conversion can be found in the program `string.dev`.

The string concatenation (+) can be applied in combination with strings or all numerical types; if necessary, the operands are first transformed into strings (according to their standard representation). At least one of the operands has to be already a string so that the operator can act as a string concatenator. In the following example a filename (e.g., `'Name5.tiff'`) is generated. For this purpose two string constants (`'Name'` and `'.tiff'`) and an integer value (the loop-index `i`) are concatenated:

```
for i := 1 to 5 by 1
  read_image (Bild, 'Name'+i+'.tiff')
endfor
```

`str(r)chr(s1,s2)` returns the index of the first (last) as a tuple occurrence of one character in `s2` in string `s1`, or `-1` if none of the characters occurs in the string.

`str(r)str(s1,s2)` returns the index of the first (last) occurrence of string s2 in string s1, or `-1` if s2 does not occur in the string.

`strlen(s)` returns the number of characters in `s`.

`s{i}` returns the character at index position `i` in `s`. The index ranges from zero to the length of the string minus 1. The result of the operator is a string of length one.

`s{i1:i2}` returns all characters from index position `i1` up to position `i2` in `s` as a string. The index ranges from zero to the length of the string minus 1.

`split(s1,s2)` divides the string `s1` into single substrings. The string is split at those positions where it contains a character from `s2`. As an example the result of

```
split('/usr/image:/usr/proj/image',':')
```

consists of the two strings

```
['/usr/image','/usr/proj/image']
```

t < t	less than
t > t	greater than
t <= t	less or equal
t >= t	greater or equal
t = t	equal
t # t	not equal

Table 3.12: Comparison operators.

3.5.8 Comparison Operators

In HDevelop, the comparison operators are defined not only on atomic values, but also on tuples with an arbitrary number of elements. They always return values of type `boolean`. Table 3.12 shows all comparison operators.

`t = t` and `t # t` are defined on all types. Two tuples are equal (`true`), if they have the same length and all the data items on each index position are equal. If the operands have different types (`integer` and `real`), the integer values are first transformed into `real` numbers. Values of type `string` cannot be mixed up with numbers, i.e., `string` values are considered to be not equal to values of other types.

1st Operand	2nd Operand	Operation	Result
1	1.0	=	true
[]	[]	=	true
''	[]	=	false
[1,'2']	[1,2]	=	false
[1,2,3]	[1,2]	=	false
[4711,'Hugo']	[4711,'Hugo']	=	true
'Hugo'	'hugo'	=	false
2	1	>	true
2	1.0	>	true
[5,4,1]	[5,4]	>	true
[2,1]	[2,0]	>	true
true	false	>	true
'Hugo'	'hugo'	<	true

Table 3.13: Examples for the comparison of tuples.

The four comparison operators compute the lexicographic order of tuples. On equal index positions the types must be identical, however, values of type `integer`, `real` and `boolean` are adapted automatically. The lexicographic order applies to strings, and the `boolean` `false` is considered to be smaller than the boolean `true` (`false < true`). In the program `compare.dev` you can find examples for the comparison operators.

`not l`	negation
`l and l`	logical 'and'
`l or l`	logical 'or'
`l xor l`	logical 'xor'

Table 3.14: Boolean operators.

3.5.9 Boolean Operators

The boolean operators `and`, `or`, `xor` and `not` are defined only for tuples of length 1. `l and l` is set to `true` (1) if both operands are `true` (1), whereas `l or l` returns `true` (1) if at least one of the operands is `true` (1). `l xor l` return `true` (1) if exactly one of both operands is true. `not l` returns `true` (1) if the input is `false` (0), and `false` (0), if the input is `true` (1).

3.5.10 Trigonometric Functions

All these functions work on tuples of numbers as arguments. The input can either be of type `integer` or `real`. However, the resulting type will be of type `real`. The functions are applied to all tuple values and the resulting tuple has the same length as the input tuple. For `atan2` the two input tuples have to be of equal length. Table 3.15 shows the provided trigonometric functions. For the trigonometric functions

`sin(a)`	sine of `a`
`cos(a)`	cosine of `a`
`tan(a)`	tangent of `a`
`asin(a)`	arc sine of `a` in the interval $[-\pi/2, \pi/2]$, `a` $\in [-1, 1]$
`acos(a)`	arc cosine `a` in the interval $[-\pi/2, \pi/2]$, `a` $\in [-1, 1]$
`atan(a)`	arc tangent `a` in the interval $[-\pi/2, \pi/2]$, `a` $\in [-1, 1]$
`atan2(a,b)`	arc tangent `a`/`b` in the interval $[-\pi, \pi]$
`sinh(a)`	hyperbolic sine of `a`
`cosh(a)`	hyperbolic cosine of `a`
`tanh(a)`	hyperbolic tangent of `a`

Table 3.15: Trigonometric functions.

the angle is specified in radians.

3.5.11 Exponential Functions

All these functions work on tuples of numbers as arguments. The input can either be of type `integer` or `real`. However, the resulting type will be of type `real`. The functions are applied to all tuple values and the resulting tuple has the same length as the input tuple. For `pow` and `ldexp` the two input tuples have to be of equal length. Table 3.16 shows the provided exponential functions.

exp(a)	exponential function $e^{\mathtt{a}}$
log(a)	natural logarithm $\ln(\mathtt{a})$, $\mathtt{a} > 0$
log10(a)	decadic logarithm, $\log_{10}(\mathtt{a})$, $\mathtt{a} > 0$
pow(a1,a2)	$\mathtt{a1}^{\mathtt{a2}}$
ldexp(a1,a2)	$\mathtt{a1} \cdot 2^{\mathtt{a2}}$

Table 3.16: Exponential functions.

3.5.12 Numerical Functions

The functions `min` and `max` select the minimum and the maximum values of the tuple values. All values either have to be of type `string`, or `integer/real`. It is not allowed to mix strings with numerical values. The resulting value will be of type `real`, if at least one of the elements is of type `real`. If all elements are of type `integer` the resulting value will also be of type `integer`. The same applies to the function `sum` that determines the sum of all values. If the input arguments are strings, string concatenation will be used instead of addition.

min(t)	minimum value of the tuple
min2(t1,t2)	element-wise minimum of two tuples
max(t)	maximum value of the tuple
max2(t1,t2)	element-wise maximum of two tuples
sum(t)	sum of all elements of the tuple or string concatenation
mean(a)	mean value
deviation(a)	standard deviation
sqrt(a)	square root $\sqrt{\mathtt{a}}$
deg(a)	convert radians to degrees
rad(a)	convert degrees to radians
real(a)	convert `integer` to `real`
int(a)	truncate `real` to `integer`
round(a)	convert `real` to `integer`
abs(a)	absolute value of `a` (`integer` or `real`)
fabs(a)	absolute value of `a` (always `real`)
ceil(a)	smallest integer value not smaller than `a`
floor(a)	largest integer value not greater than `a`
fmod(a1,a2)	fractional part of `a1`/`a2`, with the same sign as `a1`
sgn(a)	element-wise sign of a tuple

Table 3.17: Numerical functions.

The functions `sqrt`, `mean`, `deviation`, `deg`, `rad`, `fabs`, `ceil`, `floor` and `fmod` can work with `integer` and `real`; the result is always of type `real`.

The function `mean` calculates the mean value and `deviation` the standard deviation of numbers. `sqrt` calculates the square root of a number.

`deg` and `rad` convert numbers from radians to degrees and from degrees to radians, respectively.

The function `round` always returns an `integer` value and the function `abs` always returns the absolute value that is of the same type as the input value.

`real` converts an `integer` to a `real`. For `real` as input it returns the input.

`round` converts a `real` to an `integer` and rounds the value. For `integer` it returns the input.

The following example (filename: `euclid_distance.dev`) shows the use of some numerical functions:

```
V1 := [18.8,132.4,33,19.3]
V2 := [233.23,32.786,234.4224,63.33]
Diff := V1 - V2
Distance := sqrt(sum(Diff * Diff))
Dotvalue := sum(V1 * V2)
```

First, the Euclidian distance of the two vectors `V1` and `V2` is computed, by using the formula:

$$d = \sqrt{\sum_i (V1_i - V2_i)^2}$$

The difference and the multiplication (square) are successively applied to each element of both vectors. Afterwards `sum` computes the sum of the squares. Then the square root of the sum is calculated. After that the dot product of `V1` and `V2` is determined by the formula:

$$\langle V1, V2 \rangle = \sum_i (V1_i * V2_i)$$

3.5.13 Miscellaneous Functions

`sort(t)`	sorting in increasing order
`sort_index(t)`	return index instead of values
`inverse(t)`	reverse the order of the values
`is_number(v)`	test if value is a number
`number(v)`	convert `string` to a number
`environment(s)`	value of an environment variable
`ord(a)`	ASCII number of a character
`chr(a)`	convert an ASCII number to a character
`ords(s)`	ASCII number of a tuple of strings
`chrt(i)`	convert a tuple of integers into a string
`rand(a)`	create random numbers

Table 3.18: Miscellaneous functions.

`sort` sorts the tuple values in ascending order, that means, that the first value of the resulting tuple is the smallest one. But again: strings must not be mixed up with numbers. `sort_index` sorts the tuple values in ascending order, but in contrast to `sort` it returns the index positions (0..) of the sorted values.

The function `inverse` reverses the order of the tuple values. Both `sort` and `inverse` are the identity operation, if the input is empty, if the tuple is of length 1, or if the tuple contains only one value in all positions, e.g., [1,1,...,1].

`is_number` returns `true` for variables of the type `integer` or `real` and for variables of the type `string` that represent a number.

The function `number` converts a `string` representing a number to an `integer` or a `real` depending on the type of the number. Note that strings starting with `0x` are interpreted as hexadecimal numbers, and strings starting with `0` as octal numbers; for example, the string `'20'` is converted to the integer `20`, `'020'` to `16`, and `'0x20'` to `32`.
If called with a `string` that does not represent a number or with a variable of the type `integer` or `real`, `number` returns a copy of the input.

`environment` returns the value of an environment variable. Input is the name of the environment variable as a string.

`ord` gives the ASCII number of a character as an `integer`. `chr` converts an ASCII number to a character.

`ords` converts a tuple of strings into a tuple of (ASCII) integers. `chrt` converts a tuple of integers into a string.

3.5.14 Operator Precedence

Table 3.19 shows the precedence of the operators for control data. Some operations (like functions, `| |`, `t[]`, etc.) are left out, because they mark their arguments clearly.

`or`, `xor`, `bor`, `bxor`
`and`, `band`
`#`, `=`
`<=`, `>=`, `<`, `>`
`+`, `-`
`/`, `*`, `%`
`-` (unary minus), `not`
`$`

Table 3.19: Operator precedence (increasing from top to bottom).

3.6 Reserved Words

The strings shown in table 3.20 are reserved words and their usage is strictly limited to their predefined meaning. They cannot be used as variables.

true	false	and	or
xor	bor	bxor	chr
ord	chrt	ords	band
bnot	not	sum	sin
cos	tan	asin	acos
atan	sinh	cosh	tanh
exp	log	log10	ceil
floor	atan2	pow	fabs
abs	fmod	ldexp	round
deg	rad	min	min2
max	max2	sgn	rand
sort	inverse	strlen	strchr
strrchr	strstr	strrstr	split
environment	is_number	number	real
int	lsh	rsh	deviation
mean	subset	uniq	find
sqrt	gen_tuple_const	H_MSG_TRUE	H_MSG_FALSE
H_MSG_FAIL	H_MSG_VOID		

Table 3.20: Reserved words.

3.7 Control Structures

HDevelop provides the following constructs to structure programs:

`if` The simplest control structure is `if`. The condition contains a boolean expression. If the condition is true, the body is executed. Otherwise the execution is continued at the first expression or operator call that follows the word `endif`.

```
if (<Condition>)
  ...
endif
```

`ifelse` Another simple control structure is the condition with alternative. If the condition is true, all expressions and calls between the head and the word `endif` are performed. If the condition is false the part between `else` and `endif` is executed. Note that the operator is called `ifelse` and it is displayed as `if` in the program text area.

```
if (<Condition>)
  ...
else
  ...
endif
```

`while` The `while` loop has a boolean expression as the conditional part. As long as it is `true`, the body of the loop is performed. In order to enter the loop, the condition has to be true in the first place. The loop can be terminated immediately with the instruction `break` (see below).

```
while (<Condition>)
 ...
endwhile
```

In section 5.3 on page 124 you can find an example for using the `while` loop.

`for` The `for` loop is controlled by a start and termination value and an incrementation value that determines the number of loop steps. These values may also be expressions which are evaluated immediately before the loop is entered. The expressions may be of type `integer` or of type `real`. If all input values are of type `integer` the loop variable will also be of type `integer`. In all other cases the loop variable will be of type `real`.

If the start value is less or equal to the termination value, the starting value is assigned to the loop index and the body of the loop is entered. If the increment is less than zero the loop is entered if the start value is larger or equal to the end value. Each time the body is executed, the loop index is incremented by the incrementation value. If the loop index is equal to the termination value, the body of the loop is performed for the last time. If the loop index is larger than the termination value the body will not be executed any longer.[3] Please note, that the loop index does not need to become equal to the termination value in order terminate the loop. The loop index is set to the termination value when the loop is being left.

The loop can be terminated immediately with the instruction `break` (see below).

Please note, that the expressions for start and termination value are evaluated only once when *entering the loop*. A modification of a variable that appears within these expressions has no influence on the termination of the loop. The same applies to the modifications of the loop index. It also has no influence on the termination. The loop value is assigned to the correct value each time the `for` operator is executed. For more details, see section 4.4.3 on page 115 on the code generation of `for` loops.

If the `for` loop is left too early (e.g., if you press `Stop` and set the `PC`) and the loop is entered again, the expressions will be evaluated, as if the loop were entered for the first time.

```
for <loop value> := <Start> to <End> by <Increment>
  ...
endfor
```

In the following example the sine from 0 up to 6π is computed and printed in to the graphical window (filename: `sine.dev`):

[3] For negative increment values the loop is terminated if the loop index is less than the termination value.

```
old_x := 0
old_y := 0
dev_set_color ('red')
dev_set_part(0, 0, 511, 511)
for x := 1 to 511 by 1
    y := sin(x / 511.0 * 2 * 3.1416 * 3) * 255
    disp_line (WindowID, -old_y+256, old_x, -y+256, x)
    old_x := x
    old_y := y
endfor
```

In this example the assumption is made that the window is of size 512 × 512. The drawing is always done from the most recently evaluated point to the current point.

Further examples on how to use the `for` loop can be found in section 5.8 on page 135 and section 5.9 on page 137.

`break` The instruction `break` enables you to exit `for` and `while` loops. The program is then continued at the next line after the end of the loop.

A typical use of the instruction `break` is to terminate a `for` loop as soon as a certain condition becomes true, e.g., as in the following example:

```
Number := |Regions|
AllRegionsValid := 1
* check whether all regions have an area <= 30
for i := 1 to Number by 1
  ObjectSelected := Regions[i]
  area_center (ObjectSelected, Area, Row, Column)
  if (Area > 30)
    AllRegionsValid := 0
    break ()
  endif
endfor
```

In the following example, the instruction `break` is used to terminate an (infinite) `while` loop as soon as one clicks into the Graphics Window:

```
while (1)
  grab_image (Image, FGHandle)
  dev_error_var (Error, 1)
  dev_set_check ('~give_error')
  get_mposition (WindowHandle, R, C, Button)
  dev_error_var (Error, 0)
  dev_set_check ('give_error')
  if ((Error = H_MSG_TRUE) and (Button # 0))
    break ()
  endif
endwhile
```

`stop` The `stop` construct stops the program after the operator is executed. The program can be continued by pressing the `Step` or `Run` button.

`exit` The `exit` construct *terminates* the session of HDevelop.

`return` The `return` construct returns from the current procedure call to the calling procedure. `return` has no effect in case the current procedure is the main procedure.

3.8 Limitations

This section summarizes the restrictions of the HDevelop language:

- Maximum number of objects per parameter : 100000
- Maximum length of strings : 1024 characters
- Maximum length of a variable name : 256 characters
- Maximum length of a tuple : 1000000

Chapter 4

Code Export

The idea of code export or code generation is as follows: After developing a program according to the given requirements it has to be translated into its final environment. For this, the program is transferred into another programming language that can be compiled.

HDevelop allows to export a developed HDevelop program to the programming languages C++, Visual Basic, and C by writing the corresponding code to a file. The following sections describe the general steps of program development using this feature for the three languages C++ (section 4.1), COM / Visual Basic (section 4.2 on page 110), and C (section 4.3 on page 113), including some language-specific details of the code generation and optimization aspects.

Because HDevelop does more than just execute a HALCON program, the behavior of an exported program will differ in some points from its HDevelop counterpart. A prominent example is that in HDevelop, all results are automatically displayed, while in the exported programs you have to insert the corresponding display operators explicitly. Section 4.4 on page 114 describes these differences in more detail.

4.1 Code Generation for C++

This section describes how to create a HALCON application in C++, starting from a program developed in HDevelop.

4.1.1 Basic Steps

4.1.1.1 Program Export

The first step is to export the program using the menu `File` ▷ `Save As`. Here, select the language (C++) and save it to a file. In UNIX you specify the language by giving the file the extension "`.cpp`". A file will be created that contains the HDevelop program as C++ source code. For every HDevelop procedure except the main procedure, the exported file contains a C++ procedure with the corresponding name.

Iconic input and output parameters of a procedure are declared as `Hobject` and `Hobject*`, respectively, while control input and output parameters are declared as `HTuple` and `HTuple*`, respectively. All procedures are declared at the beginning of the file. The program body of the HDevelop main procedure is contained in a procedure `action()` which is called in the function `main()`. `action()` and `main()` can be excluded from compilation by inserting the instruction `#define NO_EXPORT_MAIN` at the appropriate position in the application. This can be useful if you want to integrate exported HDevelop code into your application through specific procedure interfaces. In that case, there is typically no need to export the main procedure, which was probably used only for testing the functionality implemented in the corresponding 'real' procedures.

Besides the program code, the file contains all necessary `#include` instructions. All local variables (iconic as well as control) are declared in the corresponding procedures. Iconic variables belong to the class `Hobject` and all other variables belong to `HTuple`.

4.1.1.2 Compiling and Linking in Windows NT/2000/XP Environments

The next step is to compile and link this new program. In the Windows environment, Visual C++ is used for the compiling and linking. Example projects can be found in the directory `%HALCONROOT%\examples\cpp\i586-nt4`.

If you want to use Parallel HALCON, you have to include the libraries `parhalcon.lib/.dll` and `parhalconcpp.lib/.dll` instead of `halcon.lib/.dll` and `halconcpp.lib/.dll` in your project (see the Programmer's Guide, chapter 6 on page 55, for more details).

4.1.1.3 Compiling and Linking in UNIX Environments

To compile and link the new program (called, e.g., `test.cpp`) under UNIX, you can use the example `makefile`, which can be found in the directory `$HALCONROOT/examples/cpp`, by calling

```
make PROG=test
```

Alternatively, you can set the variable `PROG` in `makefile` to `test` and then just type `make`.

You can link the program to the Parallel HALCON libraries by calling

```
make parallel PROG=test
```

or just type `make parallel` if you set the variable `PROG` as described above.

For more details see the Programmer's Guide, chapter 6 on page 55.

4.1.2 Optimization

Optimization might be necessary for variables of class `HTuple`. This kind of optimization can either be done in HDevelop or in the generated C++ code. In most cases optimization is not necessary if you program according to the following rules.

1. Using the tuple concatenation, it is more efficient to extend a tuple at the “right” side, like:

```
T := [T,New]
```

because this can the transformed to

```
T.Append(New);
```

in C++ and requires no creation of a new tuple, whereas

```
T := [New,T]
```

which is translated into

```
T = New.Append(T);
```

would need the creation of a new tuple.

2. Another good way to modify a tuple is the operator `insert` (see section 3.5.2 on page 86). In this case HDevelop code like

```
T[i] := New
```

can directly be translated into the efficient and similar looking code

```
T[i] = New;
```


4.1.3 Used Classes

There are only two classes that are used: `HTuple` for control parameters and `Hobject` for iconic data. There is no need for other classes as long as the program has the same functionality as in HDevelop. When editing a generated program you are free to use any of the classes of HALCON/C++ to extend the functionality.

4.1.4 Limitations and Troubleshooting

Besides the restrictions mentioned in this section and in section 4.4 on page 114, please also check the description of the HDevelop operators in section 2.3.8.2 on page 51.

4.1.4.1 Exception Handling

In HDevelop, every exception normally causes the program to stop and report an error message in a dialog window. This might not be useful in C++. In addition, there are different default behaviors concerning the result state of operators.

Messages

In HALCON/C++ only severe errors cause an exception handling which terminates the program and prints an error message. This might cause problems with minor errors, so called *messages* in HALCON. These messages are handled as return values of the operators and can have the following values, which are also available in HDevelop as constants:

```
H_MSG_TRUE
H_MSG_FALSE
H_MSG_FAIL
H_MSG_VOID
```

One of these messages is always returned indicating the status of the operator. Normally, the result is `H_MSG_TRUE`. Some operators return `H_MSG_FAIL` like `read_image` or `read_region` to indicate that they could not open a file or there was no permission to read it. In this case the programmer has to check the return value and apply some adequate action. If the message `H_MSG_FALSE` is ignored, errors like

```
Halcon Error #4056: Image data management: object-ID is NULL
```

will happen in successive operators, because the predecessor operator did not calculate an appropriate value.

Errors

In the case of hard errors (i.e., no message as described above) the program stops with an error message. To prevent this behavior the HDevelop operators `dev_error_var` and `dev_set_check` can be used to control the exception handling in the application. This works similarly in HDevelop and C++. One difference is caused by the dynamic evaluation of `dev_error_var` in HDevelop. This means that each time the operator is executed (e.g., in a loop) the use of the error variable might change. In contrast to this, in C++ special code is added to store the return values of operators. This code will therefore be static and cannot change during program execution. To understand how the code generation works let us have a look at a short example. Here at first the HDevelop program:

```
dev_set_check('~give_error')
dev_error_var(error,true)
threshold(image,region,100,255)
dev_error_var(error,false)
if (error # H_MSG_TRUE)
  write_string(WindowId,'error number = ' + error)
  exit()
endif
dev_set_check('give_error')
```

This program will be translated into

```
HTuple error;
::set_check("~give_error");
error = ::threshold(image,&region,100,255);
if (error != 2)
{
  ::write_string(WindowId,HTuple("error number = ") + HTuple(error));
  exit(1);
}
::set_check("give_error");
```

As can be seen, the operator `dev_error_var` is eliminated and replaced by the use of the error variable later on.

The points mentioned above might cause these two problems:

- If the second parameter of `dev_error_var` cannot be derived from the program (because no constant `false` or `true` are used but expressions, the value will be interpreted as `true`, that means: "start to use the variable". To avoid confusion use only the constants `false` or `true` as values for the second parameter.
- The usage of a variable starts after the first call of `dev_error_var(ErrVariable,true)`. In C++ this means that all successive lines (i.e., lines "below"), until the first `dev_error_var(ErrVariable,false)` will have the assignment to `ErrVariable`. This might lead to a different behavior compared with HDevelop, if `dev_error_var` is called inside a loop, because here the operators inside the loop before `dev_error_var` might also use `ErrVariable` after the second execution of the loop body. Therefore: Try not to use `dev_error_var` inside a loop. Use it right at the beginning of the program.

4.1.4.2 Compiler errors

Sometimes it happens that messages like

```
CC: "./example.cpp", line 17: bad operands for *: int * HTuple
```

or

```
CC: "./example.cpp", line 17: error ambiguous call
CC: "./example.cpp", line 17: choices of HTuple::operator *():
CC: "./example.cpp", line 17:       HTuple::operator *(const HTuple&) const;
CC: "./example.cpp", line 17:       HTuple::operator *(double) const;
CC: "./example.cpp", line 17:       HTuple::operator *(int) const;
```

are reported by the compiler. Both errors are caused by conflicting operators. In this case one either has to change the HDevelop or the C++ program. To understand how, let us look at the code which caused the errors above.[1] For the first error the C++ program would look like this:

[1] Both concrete errors shown above are hypothetical, as they would be avoided by the automatic code generation in this special case. But they are good examples for similar errors that might be caused by conflicting operators.

```
HTuple  T1,T2;
T1 = 2 * T2;
```

Because there is no operator `int * HTuple` a compiler error is given. This error can be handled in two ways:

1. Do appropriate type casting in C++: `T1 = HTuple(2) * T2;`
2. Change the order of the Operands in HDevelop and export the program again: `T1 = T2 * 2;`

Both changes will do. The first one would be used by the code generation anyway.

The second error mentioned above is caused by a similar reason. The program might look like this:

```
HTuple  T1,T2;
long    val;
T1 = T2 * val;
```

In this case `val` is a `long` variable and there is no multiplication available for the type `long` in the class `HTuple`. So again we have to change the program slightly by adding the cast Operator:

```
HTuple  T1,T2;
long    val;
T1 = T2 * HTuple(val);
```

4.2 Code Generation for Visual Basic 6

This section describes how to create a HALCON application in Visual Basic 6, starting from a program developed in HDevelop. HALCON can be used together with Visual Basic 6 based on the COM interface of HALCON. A detailed description of this interface can be found in the Programmer's Guide, part III on page 67.

4.2.1 Basic Steps

4.2.1.1 Export

The first step is to export the program using the menu `File` ▷ `Save As`. Here, select the language (`Visual Basic 6.0`) and save it to file. In UNIX you specify the language by giving the file the corresponding extension, which is "`.bas`". The result is a new file with the given name and the extension "`.bas`".

4.2.1.2 The Visual Basic 6 Template

The exported file is intended to be used together with the predefined Visual Basic 6 project that can be found in the directory

```
%HALCONROOT%\examples\vb\HDevelopTemplate
```

This project contains a form with a display window (HWindowXCtrl) and a button labeled Run. The file generated by HDevelop has to be added to this project. This is done by using the menu Project ▷ Add Module ▷ Existing and selecting the file. Now the project is ready for execution: Run the project and then press the button Run on the form, which will call the exported code.

4.2.2 Program Structure

The file created by HDevelop contains a subroutine with the corresponding name for every HDevelop procedure except the main procedure, which is contained in the subroutine action(). Iconic input and output parameters of a procedure are passed as ByVal HUntypedObjectX and ByRef HUntypedObjectX, respectively, while control input and output parameters are passed as ByVal Variant and ByRef Variant, respectively. The subroutine RunHalcon() contains a call to the subroutine action() and has a parameter Window, which is of type HWindowX. This is the link to the window on the form to which all output operations are passed. In addition, another subroutine is created with the name InitHalcon(). This subroutine applies the same initializations that HDevelop performs.

Most of the variables (iconic as well as control) are declared locally inside the corresponding subroutines. Iconic variables belong to the class HUntypedObjectX and control variables belong to Variant. The subroutine RunHalcon() has a parameter Window, which is of type HWindowX. This is the link to the window in the panel to which all output operations are passed.

Depending on the program, additional subroutines and variables are declared.

4.2.2.1 Arrays

If a single value is inserted into a Variant array, a special subroutine is called to ensure that the index is valid. If the array is too small it is resized.

4.2.2.2 Expressions

All parameter expressions inside HDevelop are translated into expressions based on the HALCON tuple operators. Therefore, an expression might look somewhat complex. In many cases these expressions can be changed to simple Visual Basic expressions. For example, TupleSub becomes a simple subtraction. To ensure that the exported program has the same effect in Visual Basic, this exchange is not applied automatically because the semantics are not always identical.

4.2.2.3 Stop

The HDevelop operator stop is translated into a subroutine in Visual Basic that creates a message box. This message box causes the program to halt until the button is pressed.

4.2.2.4 Exit

The HDevelop operator `exit` is translated into the Visual Basic routine `End`. Because this routine has no parameter, the parameters of `exit` are suppressed.

4.2.2.5 Used Classes

There are only six classes/types that are used: `Variant` for control parameters and `HUntypedObjectX` for iconic data. In addition, there is the container class `HTupleX`, which comprises all operators of HALCON processing tuples, in this case the data type `Variant`. Then, there are the classes `HWindowXCtrl` and its low-level content `HWindowX`. `HWindowXCtrl` is used inside the project for the output window and a variable of class `HWindowX` directs the output to this window. Finally, the class `HOperatorSetX` is used as a container for all HALCON operators. There is no need for other classes as long as the program has the same functionality as in HDevelop. When editing a generated program you are free to use any of the classes of HALCON/COM to extend the functionality.

4.2.3 Limitations and Troubleshooting

Besides the restrictions mentioned in this section and in section 4.4 on page 114, please also check the description of the HDevelop operators in section 2.3.8.2 on page 51.

4.2.3.1 Variable Names

In contrast to C, C++, or HDevelop, Visual Basic has many reserved words. Thus, the export adds the prefix `ho_` to all iconic and `hv_` to all control variables, respectively, in order to avoid collisions with these reserved words.

4.2.3.2 Exception Handling

In HDevelop, every exception normally causes the program to stop and report an error message in a dialog window. This might not be useful in Visual Basic. The standard way to handle this in Visual Basic is by using the `On Error Goto` command. This allows to access the reason for the exception and to continue accordingly. Thus, for HDevelop programs containing error handling (`dev_error_var`) the corresponding code is automatically included.

Please note that a call of `(dev_)set_check("~give_error")` has no influence on the operator call. The exception will *always* be raised. This is also true for messages like `H_MSG_FAIL`, which are not handled as exceptions in C++, for example.

When handling exceptions you also have to be aware that the COM interface always resets the output parameters at the beginning of the operator execution. Thus, when the exception occurs, output variables are set to `Nothing`. Therefore, you cannot use the values of variables used as output parameters of the operator causing the exception.

4.3 Code Generation for C

This section describes how to create a HALCON application in C, starting from a program developed in HDevelop.

4.3.1 Basic Steps

4.3.1.1 Program Export

The first step is to export the program using the menu `File` ▷ `Save As`. Here, select the language (C) and save it to file. In UNIX you specify the language by giving the file the extension "`.c`". A file will be created that contains the HDevelop program as C source code. For every HDevelop procedure except the main procedure, the exported file contains a C procedure with the corresponding name. Iconic input and output parameters of a procedure are declared as `Hobject` and `Hobject*`, respectively, while control input and output parameters are declared as `Htuple` and `Htuple*`, respectively. All procedures are declared at the beginning of the file. The program body of the HDevelop main procedure is contained in a procedure `action()` which is called in function `main()`. `action()` and `main()` can be excluded from compilation by inserting the instruction `#define NO_EXPORT_MAIN` at the appropriate position in the application. This can be useful if you want to integrate exported HDevelop code into your application through specific procedure interfaces. In that case, there is typically no need to export the main procedure, which was probably used only for testing the functionality implemented in the corresponding 'real' procedures.

Besides the program code, the file contains all necessary `#include` instructions. All local variables (iconic as well as control) are declared in the corresponding procedures. Iconic variables belong to the class `Hobject` and all other variables belong to `Htuple`.

Please note that in the current version the generated C code is not optimized for readability. It is output such that it always produces identical results as the HDevelop code.

4.3.1.2 Compiling and Linking in Windows NT/2000/XP Environments

The next step is to compile and link this new program. In the Windows environment, Visual C++ is used for the compiling and linking. Example projects can be found in the directory `%HALCONROOT%\examples\c\i586-nt4`.

If you want to use Parallel HALCON, you have to include the libraries `parhalcon.lib/.dll` and `parhalconc.lib/.dll` instead of `halcon.lib/.dll` and `halconc.lib/.dll` in your project (see the Programmer's Guide, chapter 15 on page 109, for more details).

4.3.1.3 Compiling and Linking in UNIX Environments

To compile and link the new program (called, e.g., `test.c`) under UNIX, you can use the example `makefile`, which can be found in the directory `$HALCONROOT/examples/c`, by calling

```
make TEST_PROG=test
```

Alternatively, you can set the variable TEST_PROG in makefile to test and then just type make.

You can link the program to the Parallel HALCON libraries by calling

```
make parallel TEST_PROG=test
```

or just type make parallel if you set the variable TEST_PROG as described above.

For more details see the Programmer's Guide, chapter 15 on page 109.

4.4 General Aspects of Code Generation

In the following, general differences in the behavior of a HDevelop program and its exported versions are described.

4.4.1 Special Comments

HDevelop comments containing the # symbol as the first character are exported as plain text statements. For example, the line

```
* #Call MsgBox("Press button to continue",vbYes,"Program stop","",1000)
```

in HDevelop will result in

```
Call MsgBox("Press button to continue",vbYes,"Program stop","",1000)
```

in Visual Basic 6. This feature can be used to integrate Visual Basic, C++, or C code into a HDevelop program.

4.4.2 Assignment

In HDevelop each time a new value is assigned to a variable its old contents are removed automatically, independent of the type of the variable. In the exported code, this is also the case for iconic objects (C++: Hobject, Visual Basic 6: HUntypedObjectX) and for the class HTuple (C++), and the type Variant (Visual Basic 6), as they all have a destructor that removes the stored data. Because C does not provide destructors, the generated C code calls the operators clear_obj and destroy_tuple to remove the content of iconic output parameters (Hobject) and control output parameters (Htuple) before each operator call.

However, problems arise if a tuple (variant) contains a *handle*, for example for a file, a window, or for OCR. In this case, the memory of the handle is automatically removed *but not the data to which it points*.

In the exported programs, this data therefore has to be removed explicitly by calling the corresponding operators `close_*` like `close_ocr` or `close_ocv`. Please insert the `close_*` operators for all handles in use

- before a new value is assigned to a handle and
- at the end of the program.

In Visual Basic, the ideal way would be to use the specific COM classes for this kind of data in combination with the member function. This exchange must be done "by hand" because the export is not able to generate appropriate code.

4.4.3 `for` Loops

HDevelop and the programming languages have different semantics for loops, which may cause confusion. Because the problems are so rare and the generated code would become very difficult to understand otherwise, the code generation ignores the different semantics. These differences are:

1. In the programming languages, you can modify the loop variable (e.g., by setting it to the end value of the condition) to terminate the loop. This can't be done in HDevelop because here the current value is stored "inside" the `for`-operator and is automatically updated when it is executed again.
2. In the programming languages, you can modify the step range if you use a variable for the increment. This is also not possible with HDevelop because the increment is stored "inside" the `for`-operator when the loop is entered.
3. The last difference concerns the value of the loop variable after exiting the loop. In the programming languages, it has the value with which the condition becomes false for the first time. In HDevelop it contains the end value, which was calculated when the loop was entered.

Looking at the mentioned points, we recommend to program according to the following rules:

1. Don't modify the loop variable or the step value inside the loop. If you need this behavior, use the `while`-loop.
2. Don't use the loop variable after the loop.

4.4.4 System Parameters

You should know that HDevelop performs some changes of system parameters of HALCON by calling the operator `set_system` (see the reference manual). This might cause the exported program not to produce identical output. If such a problem arises, you may query the system parameters by means of `get_system` in HDevelop after or while running the original HDevelop version of the program. Depending to the problem, you can now modify relevant parameters by explicitly calling the operator `set_system` in the exported program.

4.4.5 Graphics Windows

The graphics windows of HDevelop and the basic windows of the HALCON libraries (C++: class `HWindow`; Visual Basic: class `HWindowXCtrl`; C: addressed via handles) have different functionality.

- **Multiple windows**
  If you use the operator `dev_open_window` to open multiple graphics windows in HDevelop, these calls will be converted into corresponding calls of `open_window` *only for C++ and C programs*. In the export of Visual Basic programs, all window operations are suppressed, because the exported code is intended to work together with the corresponding template. If you want to use more than one window in Visual Basic, you have to modify the code and project manually.

  Note that the export of programs containing multiple windows to C++ or C might be incorrect if the button `Activate` was used during program execution. Note also that HDevelop window operations that do *not* have a window handle parameter like `dev_open_window`, `dev_close_window`, or `dev_set_line_style` are suppressed in all HDevelop procedures except the main procedure. In order to perform windows operations in HDevelop procedures aimed to be exported to C++ or C, the corresponding HALCON operators like `open_window`, `close_window`, or `set_line_style` should be used instead (in that case ignore the warning issued by HDevelop).

- **Window size**
  In exported Visual Basic programs, the size of the window on the form is predefined (512×512); thus, it will normally not fit your image size. Therefore, you must adapt the size interactively or by using the properties of the window.

- **Displaying results**
  Normally, the result of every operator is displayed in the graphics window of HDevelop. This is not the case when using an exported program. It behaves like the HDevelop program running with the option: "update window = off". We recommend to insert the operator `dev_display` in the HDevelop program at each point where you want to display data. This will not change the behavior of the HDevelop program but result in the appropriate call in the exported code.

  When generating code for C++ or C, close the default graphics window (using `dev_close_window`) and open a new one (using `dev_open_window`) *before* the first call of `dev_display` in order to assure a correct export.

- **Displaying images**
  In HDevelop, images are automatically scaled to fit the current window size. *This is not the case in exported programs*. For example, if you load and display two images of different size, the second one will appear clipped if it is larger than the first image or filled up with black areas if it is smaller. For a correct display, you must use the operator `dev_set_part` *before* displaying an image with `dev_display` as follows:

  ```
  dev_set_part (0, 0, ImageHeight-1, ImageWidth-1)
  dev_display (Image)
  ```

  In this example, `Image` is the image variable, `ImageHeight` and `ImageWidth` denote its size. You can query the size of an image with the operator `get_image_pointer1`. Please consult the HALCON Reference Manuals for more details.

  Note that the operator `dev_set_part` (and its HALCON library equivalent `set_part`) is more

commonly used for displaying (and thereby zooming) *parts* of images. By calling it with the full size of an image as shown above, you assure that the image exactly fits the window.

- **Changing display parameters**
  If you change the way how results are displayed (color, line width, etc.) in HDevelop interactively via the menu `Visualization`, these changes will not be incorporated in the exported program. We recommend to insert the corresponding `Develop` operators (e.g., `dev_set_color` or `dev_set_line_width`) in the HDevelop program explicitly. This will result in the appropriate call (`set_color`, `set_line_width`, etc.) in the exported code.

Chapter 5

Program Examples

This chapter contains examples that illustrate how to program with HDevelop. To understand the examples you should have a basic knowledge of image analysis.

The user interface is described in section 1.3 on page 2 and chapter 2 on page 11. Language details are explained in chapter 3 on page 81. The examples of this chapter are also available as program code in the directory

```
%HALCONROOT%\examples\hdevelop\Manuals\HDevelop
```

To experiment with these examples we recommend to create a private copy in your working directory.

More detailed information on HALCON operators is available in the reference manuals.

5.1 Stamp Segmentation

File name: `stamps.dev`
The first example performs a document analysis task. Figure 5.1 shows a part of a stamp catalog page. It contains two types of information about stamps: a graphical presentation and a textual description of the stamp.

In this example you have to transform the textual information into a representation that can be processed by a computer with little effort. You might use an OCR program for this task, but you will soon recognize that most of the available products create many errors due to the graphical presentation of the stamps. Thus another task has to be preprocessed: the elimination of all stamps (i.e., changing stamps to the gray value of the paper). After this preprocessing it is possible to process the remaining text using an OCR program.

When creating an application to solve this kind of problem, it is helpful to describe characteristic attributes of the objects to be searched (here: stamps). This task can be solved by a novice with some experience, too. In this case, a characterization might look as follows:

Figure 5.1: Part of the page of a *Michel* catalog.

- Stamps are darker than paper.
- Stamps are connected image areas that do not overlap.
- Stamps have a minimum and maximum size.
- Stamps are rectangular.

The task would be very simple if the attribute list would directly represent the program. Unfortunately,

this is not possible due to the ambiguity of spoken language. Thus you need language constructs with a precise syntax and a semantics that are as close as possible to the informal description. Using the HDevelop syntax, an appropriate program would look like this:

```
dev_close_window ()
read_image (Catalog, 'swiss1.tiff')
get_image_pointer1 (Catalog, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width/2, Height/2, 'black', WindowID)
dev_set_part (0, 0, Height-1, Width-1)
dev_set_draw ('fill')
threshold (Catalog, Dark, 0, 110)
dev_set_colored (6)
connection (Dark, ConnectedRegions)
fill_up (ConnectedRegions, RegionFillUp)
select_shape (RegionFillUp, StampCandidates, 'area',
              'and', 10000, 200000)
select_shape (StampCandidates, Stamps,
              'compactness', 'and', 1, 1.5)
smallest_rectangle1 (Stamps, Row1, Column1, Row2, Column2)
dev_display (Catalog)
dev_set_draw ('margin')
dev_set_line_width (3)
disp_rectangle1 (WindowID, Row1, Column1, Row2, Column2)
```

Figure 5.2 shows the segmentation result.

Due to the unknown operators and unfamiliar syntax this program appears unclear to the user at first glance.

But if you look closer at the operators you will notice the direct relation to the description above.

`threshold` selects all image pixels darker than the paper.

`connection` merges all selected pixels touching each other to connected regions.

`select_shape` selects the regions with areas (attribute: `'area'`) inside a specified interval.

`smallest_rectangle1` computes each region's coordinates (row/column) of the enclosing rectangle.

Once the user is familiar with the single operators and their syntax, the transformation becomes easy. In particular, it is not important to the program whether *an image* or *a set of regions* is processed. You can handle them both in the same way. In addition, the memory management of internal data structures is transparent to the user. Thus, you do not need to bother about memory management and you can concentrate on the image analysis tasks to solve.

5.2 Capillary Vessel

File name: `vessel.dev`
The task of this example is the segmentation of a capillary vessel. In particular, you have to separate the cell area in the upper and lower part of figure 5.3 (left image) from the area in the middle of the image.

Figure 5.2: Segmentation result for stamps.

The area boundaries are very blurred and even a human viewer has difficulties recognizing them. At first glance it seems very difficult to find a segmentation criterion: There is neither a clear edge nor a significant difference between the gray values of both areas. Thus it is not very promising to use an edge operator or a threshold operation.

One solution of this problem makes use of the different textures within the areas: Cells are more textured than the part which is supplied with blood. To emphasize this difference you can use a *texture transfor-*

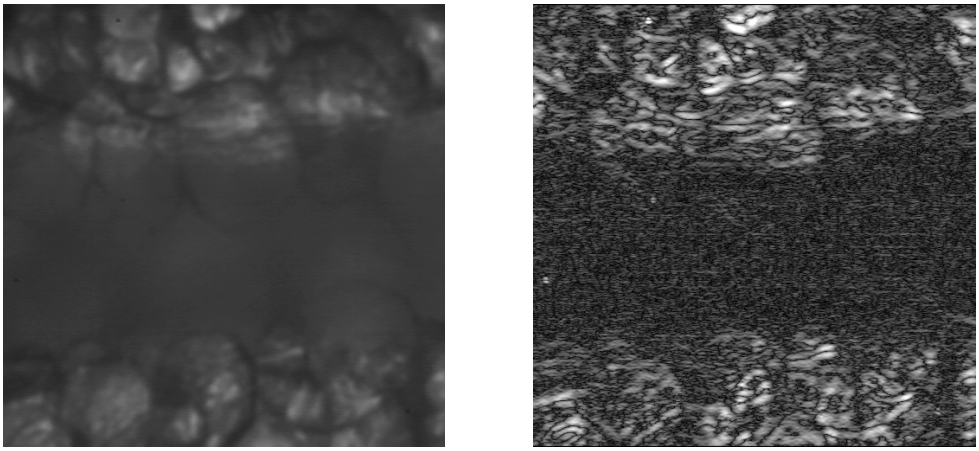

Figure 5.3: Capillary vessel (left) and texture transformation (right).

mation by Laws. Texture transformations are linear filters that intensify certain frequencies which are typical for the requested texture. The corresponding HALCON operator is `texture_laws`. You have to specify the filter size and type. Both attributes determine the frequency properties. In this program the filter `'el'` with mask size 5×5 is used. It performs a derivation in vertical direction and a smoothing in horizontal direction. Thus structures in vertical direction are intensified. You cannot directly use the computed result of `texture_laws` (see figure 5.3 right), because it is too speckled. Therefore you must generalize the texture image by a mean filter (`mean_image`). From this you obtain the so called *texture energy* (figure 5.4 left).

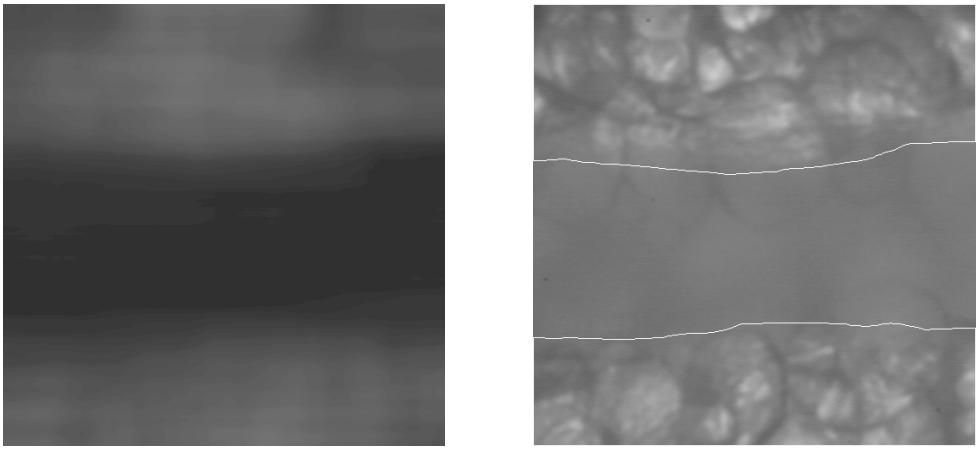

Figure 5.4: Capillary vessel texture energy (left) and segmentation (right).

The filter mask is chosen very large within this program. The mask size for the horizontal direction is 211 and 61 for the vertical direction. The asymmetry is used, because the vessel is nested in horizontal

direction. From this you obtain an image with an upper and lower part that is brighter than that in the middle.

```
read_image (Image, 'vessel')
texture_laws (Image, Texture, 'el', 5, 5)
mean_image (Texture, Energy, 211, 61)
bin_threshold (Energy, Vessel)
```

To separate these areas you just have to find the appropriate threshold. In this case — we have only two types of textures — the threshold can be found automatically. This is done by the operator `bin_threshold`, which also applies the resulting threshold and thus extracts the vessel. The right side of figure 5.4 shows the result of the segmentation.

5.3 Particles

File name: `particle.dev`
This program example processes an image that was taken from a medical application. It shows tissue particles on a carrier (figure 5.5 left).

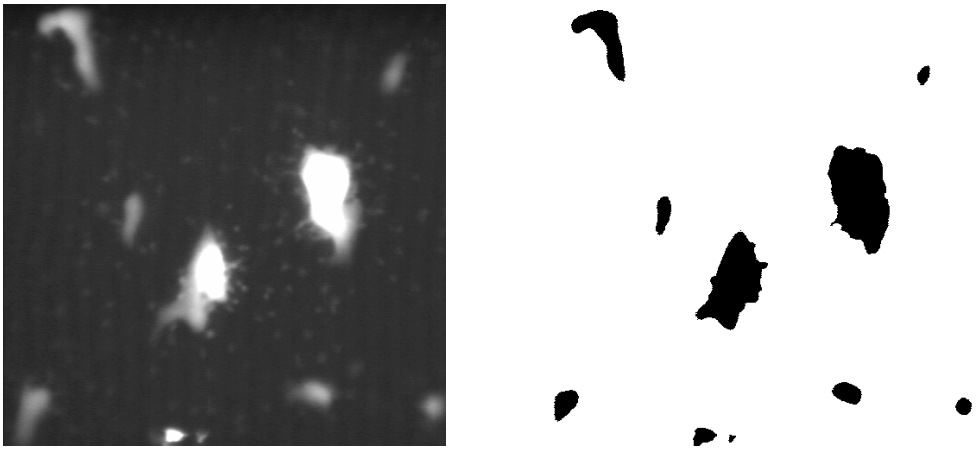

Figure 5.5: Tissue particles (left) and large objects (right).

As in many other medical applications, the existing objects have to be evaluated statistically. This means that different objects have to be extracted and classified according to their size or other attributes for example. After this, you can analyze them. An important step to solve this problem is the image segmentation that locates the relevant objects. For the statistical evaluation you may have a look at appropriate literature about statistics.

In our case there are two object classes:

- large, bright particles
- small, dark particles

The large, bright particles differ clearly from the background because of their gray values. The informal description 'brighter than the background' leads directly to the algorithmic solution using a thresholding. The only thing to decide is whether you specify the threshold automatically or empirically. In our case, a fixed threshold is completely sufficient due to the good contrast. Hence you get the following simple segmentation operator:

```
read_image (Particle, 'particle')
threshold (Particle, Large, 110, 255)
```

The variable `Large` contains all pixels whose gray values are brighter than 110. You can see the result on the right side of figure 5.5.

It is more difficult to find the small, dark particles. A first effort to specify a threshold interactively shows that there is no fixed threshold suitable to extract all particles. But if you look closer at the image you will notice that the smaller particles are much brighter than their local environment, i.e., you may specify suitable threshold values that are valid for a small image part each. Now it is easy to transform this observation into an algorithm. One way is to determine the threshold values locally (e.g., from a bar chart). Another solution might be the definition of a local environment by an $n \times n$ window. This method is used in the example. The window's mean value is used as an approximation of the background intensity. This can be done by applying a low pass filter, such as a mean filter or a Gaussian filter. The window size n defines the size of the local environment and should approximately be twice as large as the objects to search for. Since they show an average diameter of 15 pixels, a mask size of 31 is used.

The resulting pixels are specified by the comparison of the original gray values with the mean image. To reduce problems caused by noise you add a constant to the mean image (3). The appropriate program segment looks as follows:

```
mean_image (Particle, Mean, 31, 31)
dyn_threshold (Particle, Mean, Small, 3, 'light')
```

The operator `dyn_threshold` compares two images pixel by pixel. You can see the segmentation result in figure 5.6 left.

As we see, all objects have been found. Unfortunately, the edges of the large particles and several very small regions that emerged due to the noisy image material were found, too.

We first try to suppress the edges. One way is to eliminate all objects that exceed a certain maximum size. You can do this by calling:

```
connection (Small, SmallSingle)
select_shape (SmallSingle, ReallySmall, 'area', 'and', 1, 300)
```

By the same method you might also eliminate all objects which are too small (blurring). For this, you would just have to increase the minimum size with the call of `select_shape`. But if you examine the segmentation results again, you will notice that some of the resulting pixels were already extracted by the first segmentation. Thus you should search the small particles within the complement of the large ones only. To avoid the segmentation of small particles in the direct neighbourhood of the large ones, those are enlarged before building their complement. Thus we get the following modified program:

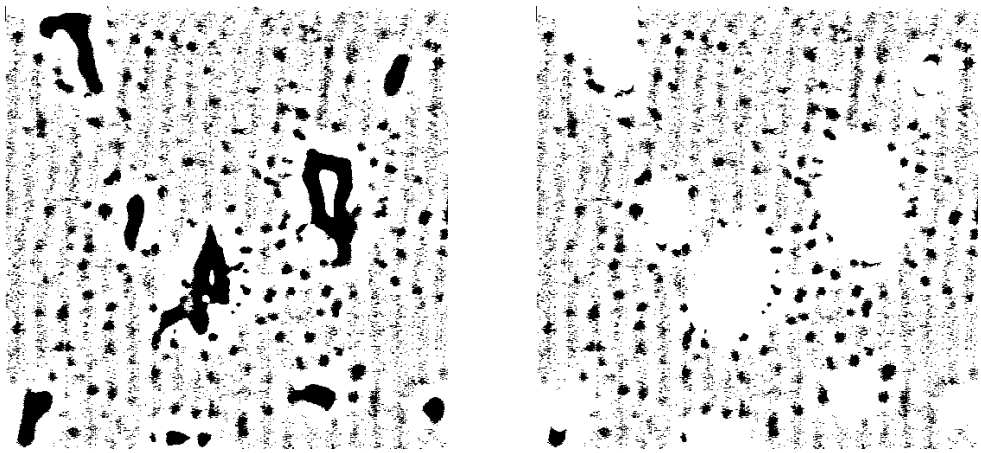

Figure 5.6: Small objects: simple (left) and advanced segmentation (right).

```
dilation_circle (Large, LargeDilation, 8.5)
complement (LargeDilation, NotLarge)
reduce_domain (Particle, NotLarge, ParticleRed)
mean_image (ParticleRed, Mean, 31, 31)
dyn_threshold (ParticleRed, Mean, Small, 3, 'light')
```

This method shows two advantages: First, the (reliable) model of the large particles can be used to extract the small ones. This increases the quality of the segmentation. Second, the processing speed is increased, as the second segmentation works only on a part of the image data. The right side of figure 5.6 shows the segmentation result.

Unfortunately, the image still contains noise. To remove it, you may either sort out noisy objects by their area as described above, or by an *opening* operation. We prefer the second method as it additionally smooths the object edges.

```
opening_circle (Small, SmallClean, 2.5)
```

Here, a circle is used as the structuring element of the opening operation. The operator preserves regions only that may at least cover a circle of radius 2.5. Smaller regions are eliminated.

Figure 5.7 shows the result of the segmentation with noise removal on the left side. The right side contains the final result.

Finally, we would like to show within this example how to select regions with the mouse interactively. At this, a loop is executed until you press the middle or right mouse button. When pressing a mouse button, the operator `get_mbutton` returns the button that was pressed and the position (coordinates) where it was pressed. This information is used to select the chosen object. In the following you see the corresponding program part:

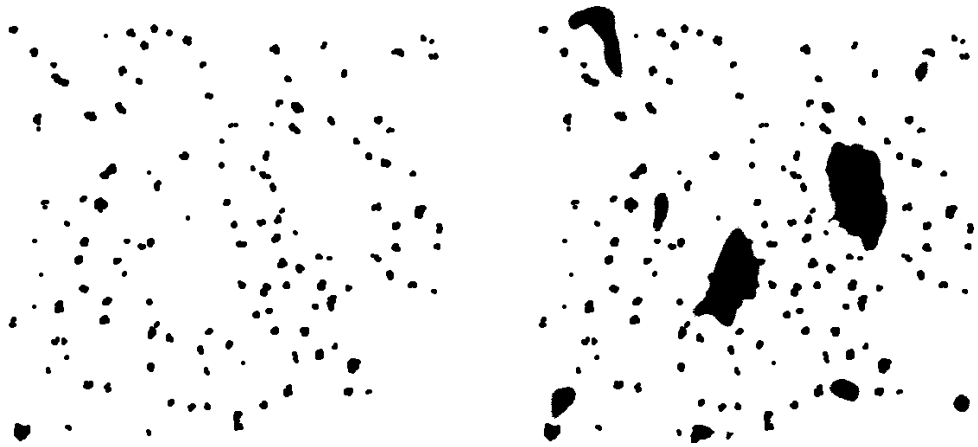

Figure 5.7: Noise-removed segmentation (left) and final result (right).

```
dev_clear_window (WindowID)
connection (SmallClean, SmallSingle)
Button := 1
dev_set_color ('red')
while (Button = 1)
  get_mbutton (WindowID, Row, Column, Button)
  select_region_point (SmallSingle, OneObject, Row, Column)
  intensity (OneObject, Particle, MeanGray, Deviation)
endwhile
```

First, the window is cleared via dev_clear_window. After that, connection calculates all connected components to allow the selection of single regions. This also displays the region components in the HDevelop window. Then you may set the drawing color (here: red) to visualize the selected regions. The loop is initialized by assigning 1 to the variable Button (1 is the code for the left mouse button). Within the loop the mouse state is queried and the chosen region is selected. As an example the mean gray value and the standard deviation are computed for each selected region. As long as you press only the left mouse button within the window the loop continues. You can terminate it by pressing any other mouse button.

5.4 Annual Rings

File name: wood.dev

Everyone knows the task to determine the age of a tree by counting its annual rings. This will now be done automatically using the example program. The first step is the segmentation of annual rings. This is quite simple as you can see them clearly as bright or dark lines. Again, the dynamic thresholding (dyn_threshold) can be used (as before during the particle segmentation in section 5.3 on page 124). To achieve a suitable threshold image you apply the mean filter (mean_image) with size 15×15 first.

The segmentation result contains many tiny regions that are no annual rings. To eliminate them you have to create the connected components (connection) and suppress all regions that are too small (select_shape). Counting the rings becomes difficult, as there might be fissures in the wood (see figure 5.8).

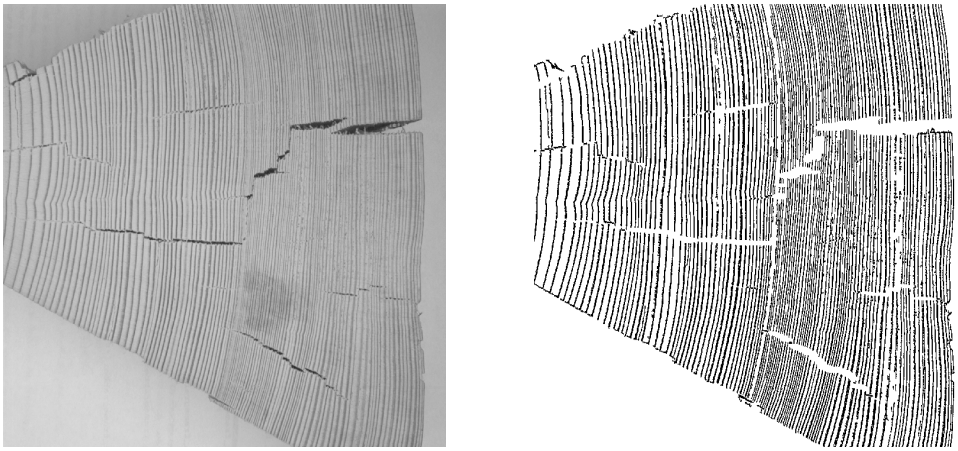

Figure 5.8: Annual rings of a tree.

Thus, we suggest the following method: You define the start and end point of a line across the annual rings using your mouse. Then the number of intersections with annual rings is counted along this line. This can be done by the following HALCON operators: The start and end points, represented by their x- and y-coordinates, are transformed into a line (gen_region_line). This line is intersected (intersection) with the annual rings (SelectedRegions). The number of the connected regions (count_obj) in this intersection is the number of annual rings. The complete program looks as follows:

```
dev_close_window ()
read_image (WoodPiece1, 'woodring')
get_image_pointer1 (WoodPiece1, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width/2, Height/2, 'black', WindowID)
mean_image (WoodPiece1, ImageMean, 9, 9)
dyn_threshold (WoodPiece1, ImageMean, Regions, 5.0, 'dark')
threshold (WoodPiece1, Dark, 0, 90)
dilation_rectangle1 (Dark, DarkDilation, 30, 7)
difference (Regions, DarkDilation, RegionBright)
connection (RegionBright, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions,
              'area', 'and', 30, 10000000)
get_mbutton (WindowID, Row1, Column1, Button1)
get_mbutton (WindowID, Row2, Column2, Button2)
gen_region_line (Line, Row1, Column1, Row2, Column2)
intersection (Line, SelectedRegions, Inters)
connection (Inters, ConnectedInters)
Number := |ConnectedInters|
```

5.5 Bonding

File name: `ball.dev`
This is the first example in the field of quality inspection. The task is to detect bonding balls. Figure 5.9 shows two typical microscope images of a *die*.

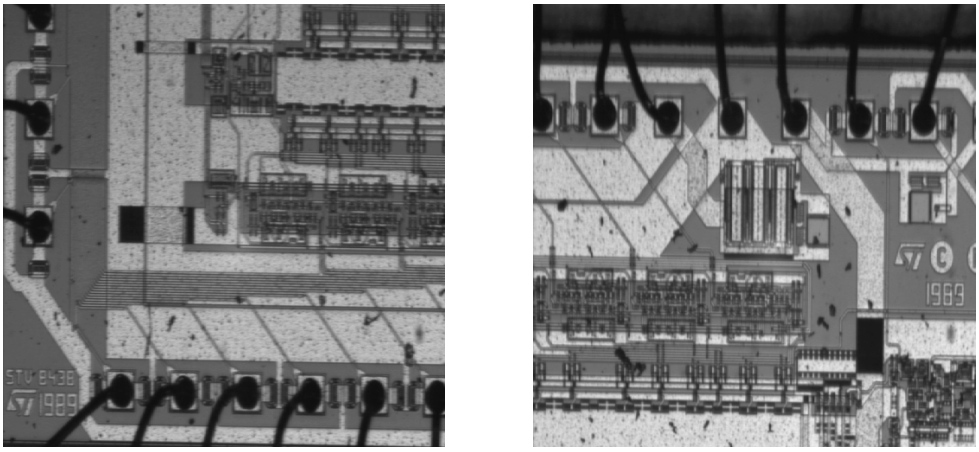

Figure 5.9: Exemplary images with bonding balls on a *die*.

The *die* border and the bonding wires appear dark. Thus you may apply a thresholding. Since the background is also dark we have to extract the *die* before doing the segmentation. The *die* is rather bright. Thus we can select the pixels by their gray values.

```
read_image (Bond, 'die3')
threshold (Bond, Bright, 120, 255)
shape_trans (Bright, Die, 'rectangle2')
```

All pixels of the *die* that got lost by the thresholding can be recovered by using a hull computation. Since the *die* is rectangular and may be slightly turned during the assembly we use the smallest enclosing rectangle as a hull.

Now you can start the segmentation of wires and bonding balls. Since only those parts of wires and balls are of interest that lie within the *die* area, you may restrict the segmentation to this region. All dark pixels within the *die* area belong to wires. Unfortunately, there are some bright reflections on the wires that are not found by the segmentation. You may fill these gaps by using `fill_up_shape`. In our case, the gaps with a certain size (1 up to 100 pixels) are filled.

```
reduce_domain (Bond, Die, DieGray)
threshold (DieGray, Wires, 0, 100)
fill_up_shape (Wires, WiresFilled, 'area', 1, 100)
opening_circle (WiresFilled, Balls, 15.5)
connection (Balls, SingleBalls)
select_shape (SingleBalls, IntermediateBalls, 'circularity',and,0.85, 1.0)
sort_region (IntermediateBalls, FinalBalls, 'FirstPoint', 'True', 'column')
smallest_circle (FinalBalls, Row, Column, Radius)
```

Since the balls are wider than the wires, you may clean this region using a simple opening. The radius (here 15.5) should correspond to the minimum size of one ball. In both images you see an erroneous segmentation that was created by a rectangular dark region. This can be suppressed by a shape segmentation. Since in practice a bonding detection would be performed only close to the anticipated positions of bonding balls. Figure 5.10 shows the results of the whole segmentation.

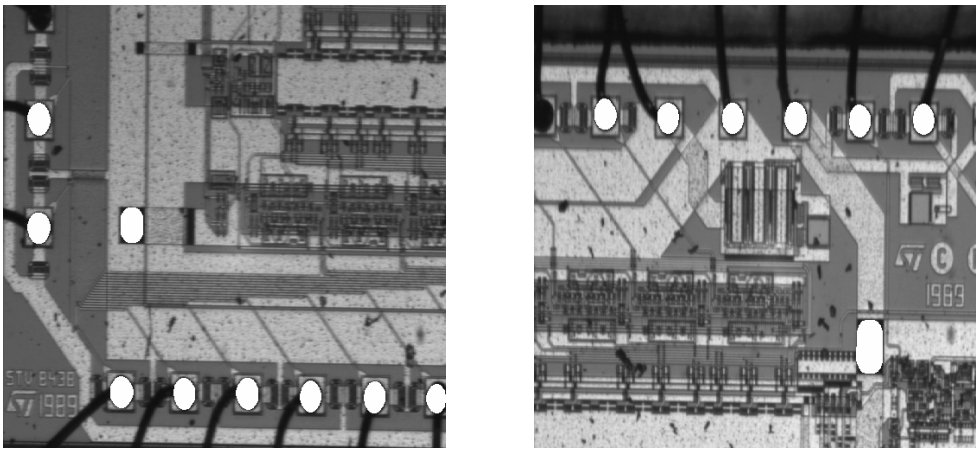

Figure 5.10: Detected bonding positions.

Balls are shown in white color. Every radius of a ball you can find in the tuple variable Radius. The number of balls within the example you can get with the absolute value of Radius.

```
NumBalls := |Radius|
Diameter := 2*Radius
MeanDiameter := sum(Diameter)/NumBalls
MinDiameter := min(Diameter)
```

Diameter, MeanDiameter and MinDiameter are some examples for calculations possible with HDevelop.

5.6 Calibration Plate

File name: `calib.dev`

This example works with the image of a calibration plate. It is used to specify the internal parameters of a CCD camera. Therefore, you have to extract the circles on the plate (see left side of figure 5.11).

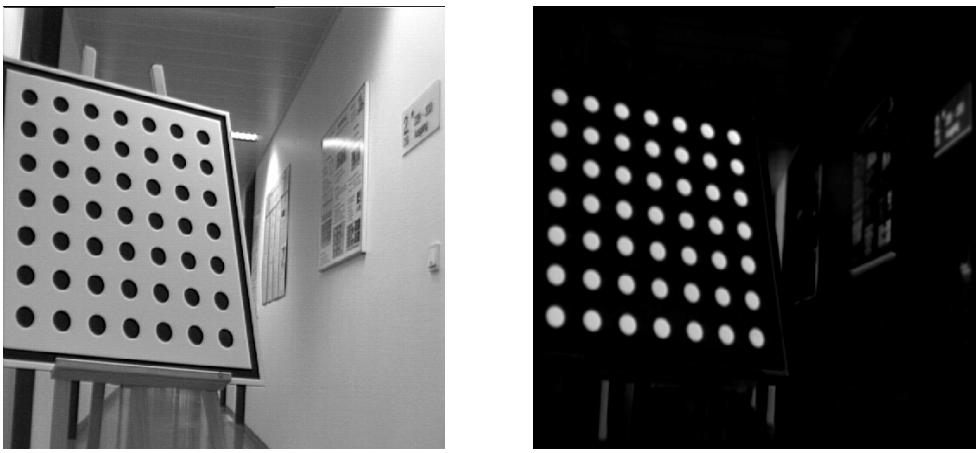

Figure 5.11: Calibration plate and `gray_inside` result.

This example describes an interesting operator. It is called `gray_inside` and is a so-called *fuzzy operator*. In this case, fuzzy means that the value of each pixel is not interpreted as gray value but as the *affiliation* to a certain class. The bigger the number (max. 1), the stronger the affiliation.[1]

By applying `gray_inside` to an image every pixel value is interpreted as the "potential energy" you have to afford to get from the pixel position to the image border. The dark pixels present valleys and the bright pixels mountains. Thus a dark region in the middle of an image is equivalent to a hole in a mountain that needs a lot of energy to be left. This is also true for the dark circles on the bright background in the image of the calibration board.

Before calling `gray_inside` you should use a smoothing filter to suppress small valleys. This reduces runtime considerably.

If you look at the operator result on the right side of figure 5.11 you will notice the circles as significant bright points. Now a simple thresholding is sufficient to extract them.

```
read_image (Caltab, 'caltab')
gauss_image (Caltab, ImageGauss, 9)
gray_inside (ImageGauss, ImageDist)
threshold (ImageDist, Bright, 110, 255)
connection (Bright, Circles)
elliptic_axis (Circles, Ra, Rb, Phi)
```

After calculating the ellipse parameters of each circle (`elliptic_axis`), you may compute the camera parameters.

[1] In HALCON the range of 0 to 1 is mapped to values of a byte image (0 to 255).

5.7 Devices

File name: `ic.dev`
This example discusses the combination of different segmentation methods. It works with an image of multiple electronic components. These differ in shape, size and arrangement. The left side of figure 5.12 shows the input image.

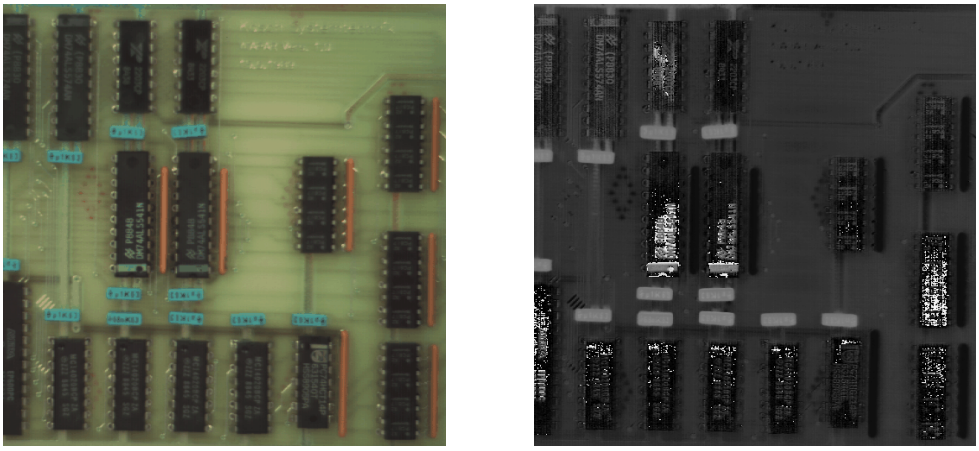

Figure 5.12: Board with electronic devices (left) and the corresponding color value image in the HSV space (right).

First you extract resistors and capacitors. This is quite simple because you have a color image and both component types have different colors. The input image consists of three channels containing the red, green, and blue channels. Since segmentation in the RGB space is difficult, you have to transform the image into the HSV space. Here the color information is stored in one single channel. The right side of figure 5.12 shows the image representation in this channel (*Hue*). Elements that are too small can be eliminated via `select_shape`. The program sequence to extract resistors and capacitors is shown below:

```
read_image (ICs, 'ic')
decompose3 (ICs, Red, Green, Blue)
trans_from_rgb (Red, Green, Blue, Hue, Saturation, Intensity)
threshold (Saturation, Colored, 100, 255)
reduce_domain (Hue, Colored, HueColored)
threshold (HueColored, Blue, 114, 137)
connection (Blue, BlueConnect)
select_shape (BlueConnect, BlueLarge, 'area', 'and', 150, 100000)
shape_trans (BlueLarge, Condensators, 'rectangle2')
threshold (HueColored, Red, 10, 19)
connection (Red, RedConnect)
select_shape (RedConnect, RedLarge, 'area', 'and', 150, 100000)
shape_trans (RedLarge, Resistors, 'rectangle2')
```

If you look closer at this program segment you will notice some obvious enhancements that can be made.

One is necessary due to the color model: The thresholding of the color image chooses all pixels with a certain color. This selection is independent of the color saturation. Thus it might happen that very bright pixels (nearly white pixels) or very dark pixels (nearly black pixels) have the same color value as the components. But you are only looking for stronger colors. For this you select all pixels first whose color is strong, i.e., all pixels with a high saturation.

The second enhancement concerns the objects' shape. As the devices are rectangular you can specify the smallest enclosing rectangle of all connected components to enhance the segments. On the left side of figure 5.13 the resulting components are marked.

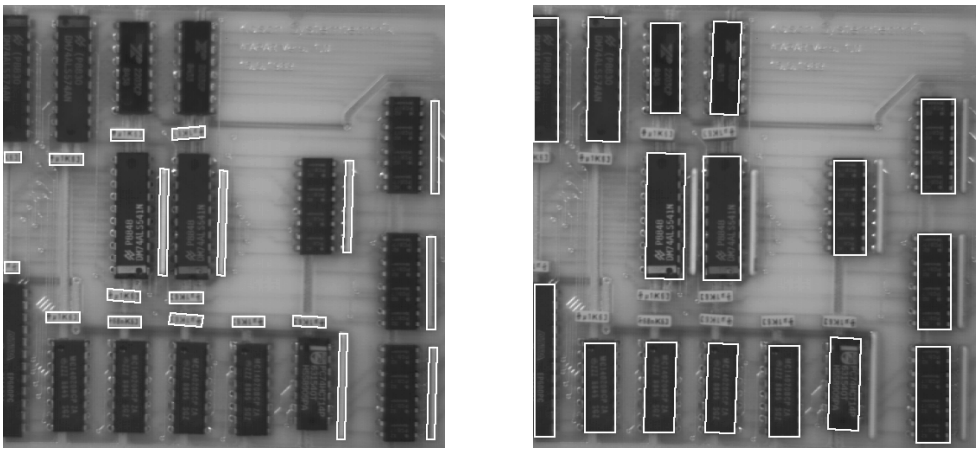

Figure 5.13: Resistors and capacitors (left) and ICs (right).

In a second step, we will search for all ICs. This seems to be easy, as they are rather large and dark. However, some problems emerge due to the bright labels that are printed across some ICs. Thus a simple thresholding alone is not sufficient. In addition you have to combine the segments belonging to one IC. This is done by examining the spatial adjacencies of the segments. A dilation is used to enlarge the regions until they overlap each other. This dilation must not be so large that different ICs are merged. Thus gaps caused by labels have to be smaller than gaps between ICs. Now you can separate the enlarged ICs in their connected components. Unfortunately, they have become too large by the dilation. Another thresholding for each connected component will detect the dark pixels of each IC. Finally, you can specify the enclosing rectangles analogously to the resistors and the capacitors (see above).

```
threshold (Intensity, Dark, 0, 50)
dilation_rectangle1 (Dark, DarkDilate, 15, 15)
connection (DarkDilate, ICLarge)
add_channels (ICLarge, Intensity, ICLargeGray)
threshold (ICLargeGray, ICsDark, 0, 50)
shape_trans (ICsDark, IC, 'rectangle2')
```

The right side of figure 5.13 shows the resulting ICs. We have to mention two aspects about the program segment above. Here the operator `add_channels` has been used instead of `reduce_domain`. This is necessary as *several* regions have to be "supplied" with gray values. The situation of previous programs

was quite different: there the number of valid pixels of *one* image has been restricted. From this follows the second point: here the operator `threshold` gets several images as input.[2] The thresholding is performed in every image. Thus you receive as many regions as input images.

Finally, the segmentation of IC contacts has to be done. They are bright and small. Thus it is easy to extract them using a dynamic thresholding (compare section 5.3 on page 124). However, several other tin elements on the board remain a problem, because they have to be distinguished from the IC contacts. This can be done by restricting the search on a *region of interest*. IC contacts may only appear either on the right or the left side of ICs. The coarse region of interest is defined by enlarging the IC regions with a following set subtraction. Then the result is resized appropriately by using another dilation. Figure 5.14 shows the operator result on the left side.

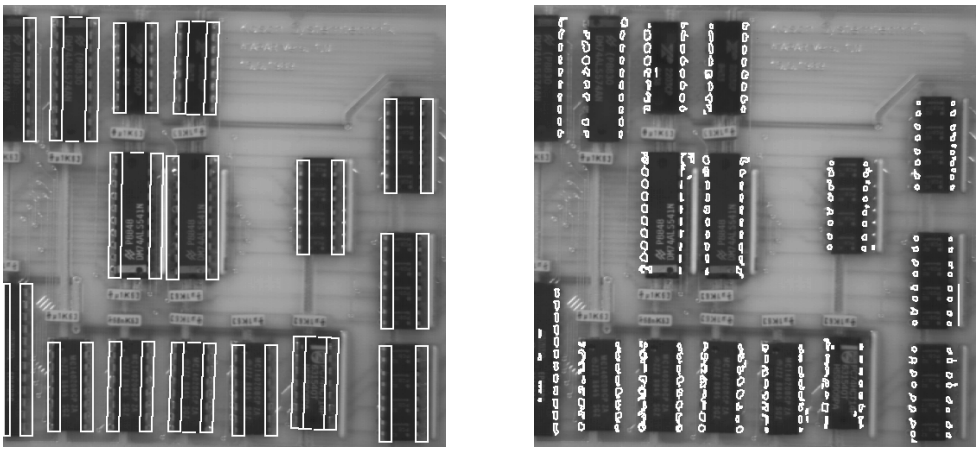

Figure 5.14: Searching regions for contacts (left) and IC contacts (right).

Now you only have to intersect the result of the thresholding with the region of interest.

```
dilation_rectangle1 (IC, ICWidth, 5, 1)
difference (ICWidth, IC, SearchingArea)
dilation_rectangle1 (SearchingArea, SearchingAreaWidth, 14, 1)
union1 (SearchingAreaWidth, SearchingAreaUnion)
reduce_domain (Intensity, SearchingAreaUnion, SearchGray)
mean_image (SearchGray, Mean, 15, 15)
dyn_threshold (SearchGray, Mean, Contacts, 5, 'light')
connection (Contacts, ContactsConnect)
fill_up (ContactsConnect, ContactsFilled)
select_shape (ContactsFilled, ContactsRes, 'area', 'and', 10, 100)
```

The result of the intersection is still not satisfying. Too many small and too many wrong regions have been found. So we have to eliminate them by using `select_shape`. Figure 5.14 shows the final result of the segmentation on the right side.

[2] One matrix is shared by several iconic objects to reduce costs of memory and computation time.

5.8 Cell Walls

File name: `wood_cells.dev`
In this example we will examine the alteration of the cell wall's proportion during a tree's growth. The input image is a microscope view of wooden cells (see figure 5.15).

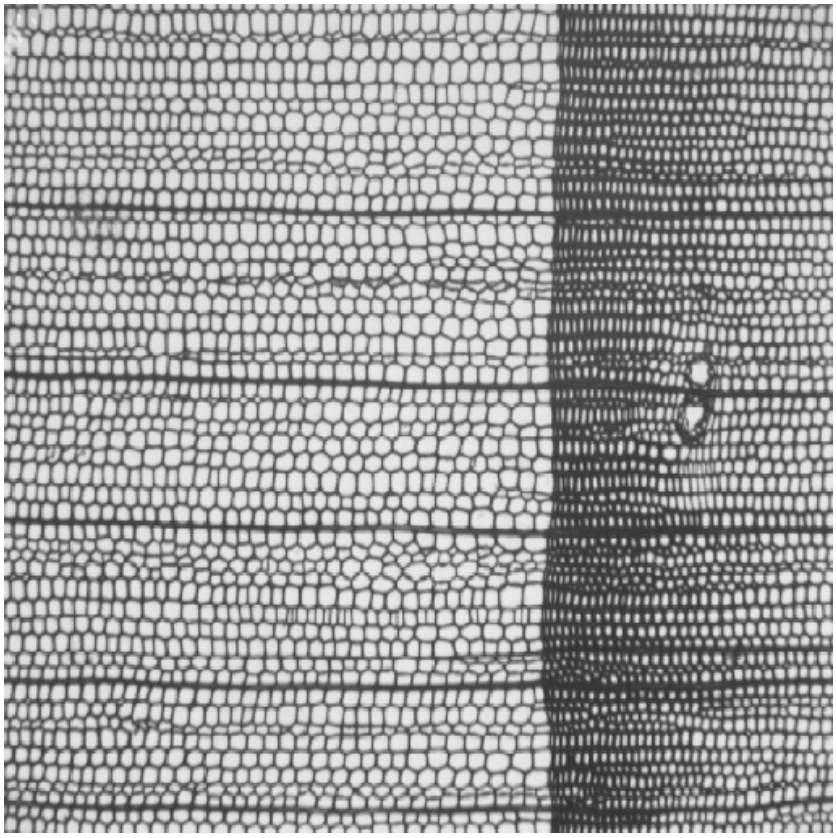

Figure 5.15: Microscope image of wooden cells.

You can clearly see the single cells and the discontinuity that is caused by the stopped growing in winter.

Extracting cell walls is simple because they are significantly darker. The remaining "difficulty" lies in the computation of the distribution in growth direction, i.e., along the image x-axis. First, we define the width of the window over which the cell distribution is computed by assigning it to the variable `X` in the program. Then we fetch the image size using `get_image_pointer1` to get the corresponding loop parameters. The broader the search range, the stronger the smoothing during the measurement.

Now the loop starts from the "left" side to compute the whole image. The ratio of the area of the cell walls and a rectangle of width `X` is computed for every value of the loop variable `i`. The number of pixels belonging to a cell wall (`Area`) is determined by `area_center`. This value is transformed to percent for the output.

```
X := 20
read_image (WoodCells1, 'woodcell')
threshold (WoodCells1, CellBorder, 0, 120)
get_image_pointer1 (WoodCells1, Pointer, Type, Width, Height)
open_file ('wood_cells.dat', 'output', FileHandle)
for i := 0 to Width-X-1 by 1
  clip_region (CellBorder, Part, 0, i, Height-1, i+X)
  area_center (Part, Area, Row, Col)
  fwrite_string (FileHandle, i + ' ' + (Area * 100.0 / (X * Height)))
  fnew_line (FileHandle)
endfor
close_file (FileHandle)
```

Figure 5.16 shows the measurement result.

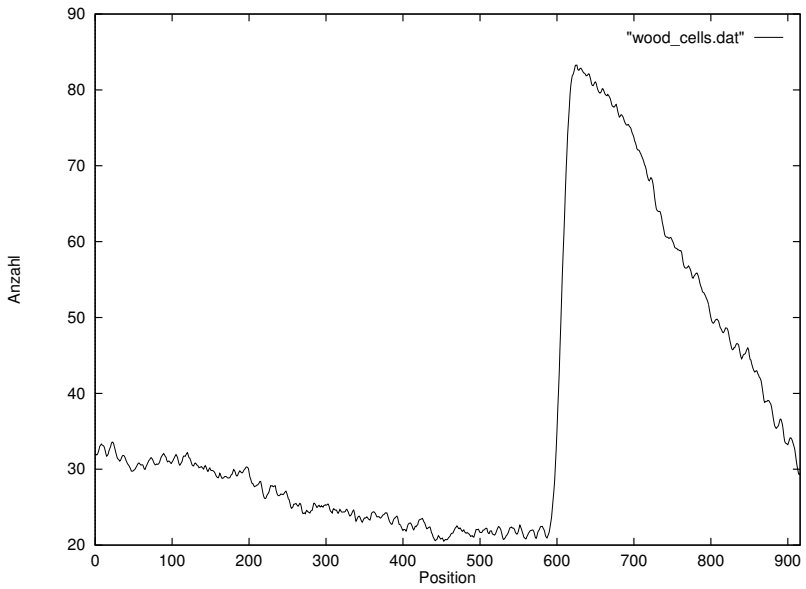

Figure 5.16: Cell wall proportion in growth direction in percent.

To allow further processing of the data (such as for plotting using *gnuplot* as in figure 5.16) it has to be written to a file. Therefore, a text file is opened first (`open_file`). Now you can write to this file by using `fwrite_string` and `fnew_line`. Note the formatting of output when using `fwrite_string`. The output text starts with the loop variable that is followed by a space character. Thus the number is transformed into a string. Finally, the proportion of the cell wall (in percent) is concatenated to the string. At this it is important that the first or second value of the expression is a string, so that the following numbers are converted into strings. Here + denotes the concatenation of characters instead of the addition of numbers.

5.9 Region Selection

File name: `eyes.dev`
This example explains how to handle single iconic objects. In contrast to numerical data, where many different functions may be executed on parameter positions (see section 3.5 on page 85), iconic objects may only be handled by using HALCON operators. The most important operators to select and combine iconic objects are shown in this example.

The task is to search the eyes of the mandrill in figure 5.17.

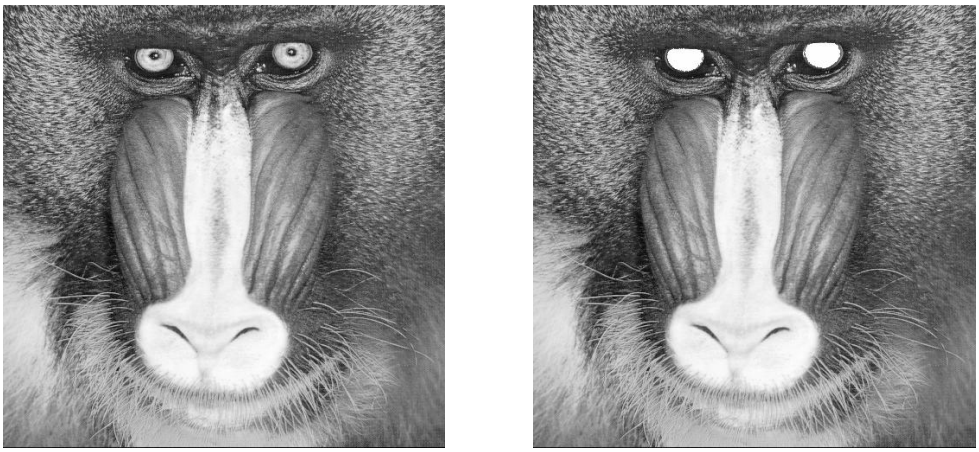

Figure 5.17: Mandrill and the detected result.

This is a simple task. First, we extract the bright parts by a thresholding. Then we have to examine the connected components according to their shape and size to select the eyes. At this, you could use the operator `select_shape` and get a fast program of five lines that processes the task. For demonstration purpose we use a kind of "low level" version instead: every region is extracted separately and examined afterwards. If it conforms to a given shape, it is added to a result variable.

```
dev_close_window ()
read_image (Image, 'monkey')
threshold (Image, Region, 128, 255)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, CompactRegions,
              'compactness', 'and', 1.5, 1.8)
Number := |CompactRegions|
Eyes := []
for i := 1 to Number by 1
  SingleSelected := CompactRegions[i]
  area_center (SingleSelected, Area, Row, Column)
  dev_set_color ('green')
  if ((Area > 500) and (Area < 50000))
    dev_set_color ('red')
    Eyes := [SingleSelected,Eyes]
  endif
endfor
```

Note that you have to specify the number of regions (count_obj) in order to run a for loop from 1 to Number. Within this loop a region is selected (select_obj) according to the loop variable i in order to evaluate its attributes. If its area is within certain bounds the region is added to variable Eyes (concat_obj). You have to specify the variable Eyes properly, as it is also used as input for concat_obj. This can be done by using empty_object that assigns no iconic object in a defined way to the variable, i.e., count_obj returns zero for it.

During the run time of the program you can see how the individual regions are selected and examined. To speed up the processing you can use the menu File ▷ Options to suppress the automatic output.

5.10 Exception Handling

File name: exception.dev
In some applications it is necessary to have explicit control over the result state of an operator. By default HDevelop stops if an operator returns a different state than H_MSG_TRUE and gives an error message. To have explicit control over the result state, two HDevelop operators are available: dev_error_var and dev_set_check. The following example shows how to use these operators.

The task is to get online information about the position of the mouse inside a graphics window and to display the gray value at this position. This can be achieved using the two operators get_mposition and get_grayval. The problem with get_mposition in HDevelop is, that it returns H_MSG_FAIL if the mouse is outside of the window to indicate that the mouse coordinates are invalid. This would lead to an interruption of the program. Therefore an explicit error handling is needed. The complete program is given below:

```
read_image (Image, 'mreut')
dev_close_window ()
dev_open_window (0, 0, -1, -1, 'black', WindowID)
dev_display (Image)
Button := 1
while (Button # 4)
  dev_error_var (Error, 1)
  dev_set_check ('~give_error')
  get_mposition (WindowID, Row, Column, Button)
  dev_error_var (Error, 0)
  dev_set_check ('give_error')
  if (Error = H_MSG_TRUE)
    get_grayval (Image, Row, Column, Grayval)
    dev_set_color ('black')
    disp_rectangle1 (WindowID, 0, 0, 22, 85)
    dev_set_color ('white')
    set_tposition (WindowID, 15, 2)
    write_string (WindowID, '('+Row+','+Column+')='+Grayval)
  endif
endwhile
```

After loading an image and opening a window we enter the loop to query the mouse position. Because the operator `get_mposition` might cause an exception we call `dev_set_check` to declare that HDevelop should not stop if an exception occurs. `dev_set_check` has to be called before and after the critical call(s). If we want to know which error occurred we have to specify the variable in which the return value will be stored. This is done by using `dev_error_var`. Now `get_mposition` can be called independent of the context. To check if the coordinates are valid, the error variable is compared to one of the constants for standard return values (like `H_MSG_TRUE` or `H_MSG_FAIL`). If the call succeeded, this coordinate is used to query the gray value of the corresponding pixel in the image, which is then displayed in the window.

5.11 Road Scene

File name: `road_signs.dev`
The computing time is a critical factor in many image analysis tasks. Thus the system has to offer features to speed up the processing. But direct hardware access must be avoided in any case. All operators should work on encapsulated data structures. To allow optimization for performance, data structures have to be used that support transparent and efficient programming. The example segmentation of a road scene demonstrates how HALCON helps to achieve this goal.

Here the task is to find the middle and border road markings of a motorway. The program is performed by a normal workstation with a processing time of maximum 20 ms per half image (video frequency) at a resolution of 512×512 pixels. In figure 5.18 you see an image of such a road sequence on the left side.

Assume that there is no specialized operator for this task. Thus, you have to make use of standard methods. The data structure used consists of a gray value image with a covering mask, i.e., the definition range.[3] All operators work only on those parts of the image data that lie within the definition range. This

[3] See the manual **Getting Started** for a short introduction to the data structures used by HDevelop.

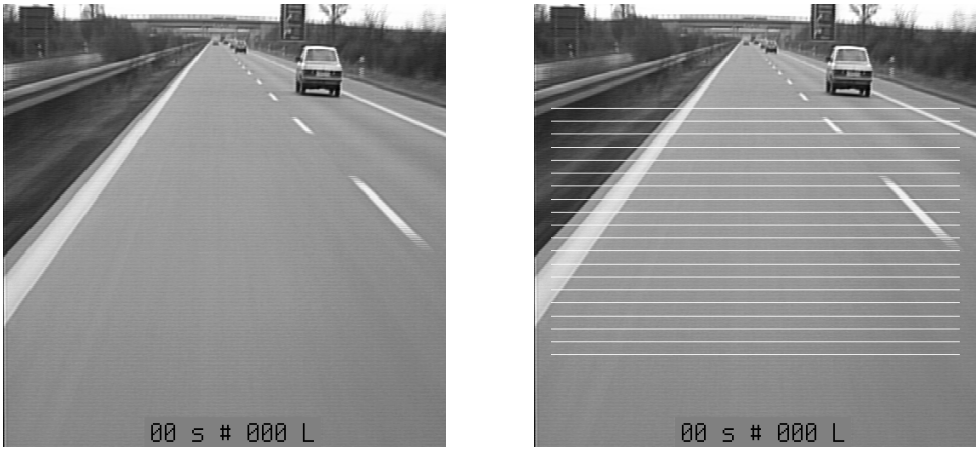

Figure 5.18: Part of an image sequence (left) and search grid for marking band (right).

can be exploited to reduce computation time.

The following assumptions on the image data help to specify a region as a search mask:

1. Road markings remain in a certain image part only.
2. Road markings have a certain minimum length in y-direction.
3. Road markings are separated by an edge from their environment.

The first two assumptions can restrict the search area enormously. To make use of this, we create a region as a grid whose line distance is determined by the minimum size of the road marking. Figure 5.18 shows the corresponding region (= line grid) on the right side.

While performing an edge filter within the grid all pixels with a high gradient are candidates on the contour of a road marking. By enlarging these pixels by the minimum diameter of the markings (dilation) with rectangle, you will get the search window shown in figure 5.19 on the left side.

Now the road markings can be easily extracted by a thresholding within the search windows. The segmentation result is shown on the right side of figure 5.19. The corresponding HDevelop program looks as follows:

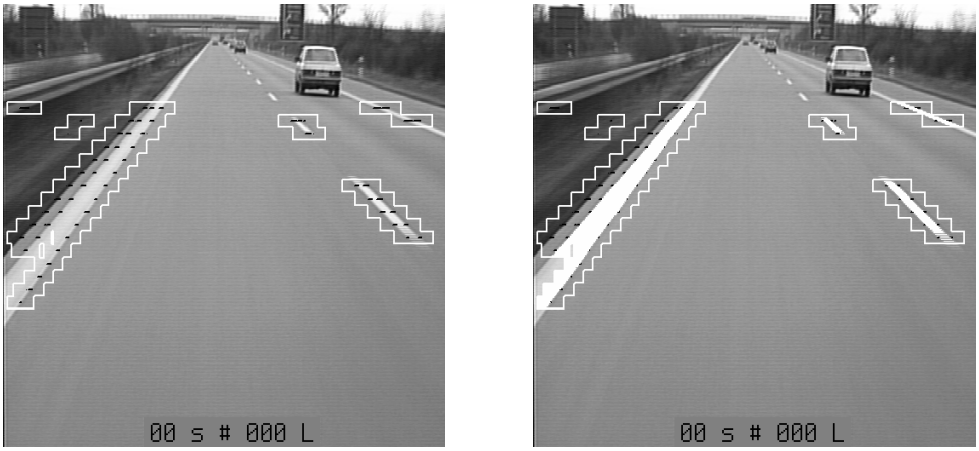

Figure 5.19: Search areas for markings (left) and segmentation (right).

```
MinSize := 30
set_system ('init_new_image', 'false')
read_image (Motorway, 'motorway')
count_seconds (Seconds1)
gen_grid_region (Grid, MinSize, MinSize, 'lines', 512, 512)
clip_region (Grid, GridRoad, 130, 10, 450, 502)
reduce_domain (Motorway, GridRoad, Mask)
sobel_amp (Mask, Gradient, 'sum_abs', 3)
threshold (Gradient, Points, 40, 255)
dilation_rectangle1 (Points, RegionDilation, MinSize, MinSize)
reduce_domain (Motorway, RegionDilation, SignsGray)
threshold (SignsGray, Signs, 190, 255)
count_seconds (Seconds2)
Time := Seconds2-Seconds1
dev_display (Signs)
```

First you create a grid by using gen_grid_region. It is reduced to the lower image half with clip_region. The operator reduce_domain creates an image containing this pattern as definition range. This image is passed to the operator sobel_amp. You obtain pixels with high gradient values using threshold. These pixels are enlarged to the region of interest (ROI) by a dilation with a rectangular mask. Within this region another thresholding is performed. Correct road markings are equivalent to bright areas in the search window (ROI).

The HALCON program needs an average of 20 ms on a standard Pentium. Notice that this is even possible under the following restrictions:

1. Only standard operators have been used.
2. Only encapsulated data structures have been used.
3. Despite optimization the program is quite comprehensible.

4. The program is very short.

This example shows that you can write efficient programs even while using complex data structures. Hence a significant reduction of development time is achieved. Furthermore, data encapsulation is a basic condition for the portability of the whole system and the user software.

Chapter 6

Tips & Tricks

This chapter contains helpful information for working with HDevelop.

6.1 Keycodes

In order to speed up the entering of values in the input fields of HDevelop (e.g., operator parameters), several keycodes are defined, which have special functions. They conform to the standards of the `Emacs` editor. This feature is only available for UNIX systems. Some of them are shown in table 6.1.

`Delete`	Delete single character at current cursor position.
`<Ctrl> a`	Move the cursor to the beginning of the line.
`<Ctrl> b`	Move cursor left one character.
`<Ctrl> d`	Analogous to `Delete`
`<Ctrl> e`	Move cursor to last character in line.
`<Ctrl> f`	Move cursor right one character.
`<Ctrl> h`	Delete single character immediately preceding current cursor position
`<Ctrl> k`	Delete all characters from current position to end of line.
`<Meta> b`	Backward to previous word.
`<Alt> b`	Backward to previous word.
`<Meta> d`	Delete from current cursor position to end of current word.
`<Alt> d`	Delete from current cursor position to end of current word.
`<Meta> f`	Forward to next word.
`<Alt> f`	Forward to next word.

Table 6.1: Keycodes for special editing functions.

6.2 Interactions During Program Execution

The interpreter of HDevelop allows some user interactions during the execution of a program. First, the stop button has to be mentioned, which is responsible for interrupting the execution of a program. When the stop button is pressed, the execution is stopped at the active HALCON operator.

Other features of the HDevelop interpreter are the possibility to display iconic variables by simply double clicking on them, and the facility to set the parameters which control the display to the appropriate values. In addition to this, it is possible to insert commands into the program text, no matter whether this makes any sense or not. Please note that interactions during the execution of HALCON application can only be used in a sensible way, if the single operators have short runtimes, because HDevelop can only react within the "gaps", that is, between the calls to the HALCON library.

Please note that neither the `PC` nor the `BP` can be set during the execution of the HALCON application.

6.3 Online Help

Online documentation is available in PDF and partly in HTML format.

To display the HTML files containing information on HALCON operators, you need a browser. It is not provided in the HALCON distribution, but nevertheless used by HDevelop. Such a tool may already be installed on your computer. Otherwise you may obtain it for free, e.g., via the Internet. One browser that is suitable for displaying HTML files is Netscape Navigator. It is a WWW browser that is able to display HTML documents. Since the reference manual for HALCON operators is also stored in HTML format, it is convenient to use a standard WWW browser. In the tool HDevelop you may call Netscape via the menu `Help ▷ HALCON Operators`. It will start Netscape with the corresponding help files (see section 2.3.11 on page 60). An alternative to Netscape is to use the Microsoft Internet Explorer.

Besides HTML, the documentation is available in PDF format as well. To display the manuals, the Adobe file viewer Acrobat Reader is included in the distribution for Windows systems. This viewer is not activated from HDevelop, but has to be started from the Windows start menu.

6.4 Warning and Error Windows

Warning and error windows are popups that make the user aware of user errors. Usually, they interrupt the faulty actions with a description of the error. For this purpose information about the kind of the error is determined during the execution. Figure 6.1 shows an example of an error window.

6.5 Restrictions

Not every HALCON operator that is available in HALCON/C or HALCON/C++ can or should be used in HDevelop. There are two reasons for this. On the one hand, the HALCON system is influenced by

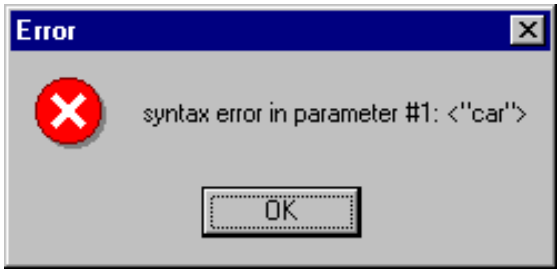

Figure 6.1: Example for an error window.

HDevelop so deeply that some operators don't behave like in a normal user program. Usually this concerns the graphical operators like `set_color`. For this class of operators, specific versions for HDevelop are available, e.g., `dev_set_color`.

On the other hand, some low-level operators exist (like `reset_obj_db` or `clear_obj`) that will bring HDevelop "out of balance."

Not the whole functionality of HDevelop can be transferred to a C++-program because the graphics windows of HDevelop are more comfortable than the simple HALCON windows.

However, the points described above are very special and will not bother the normal user because the appropriate functions can be found in both working environments. If you use `dev_set_color` in HDevelop, for example, you would use `set_color` as its counterpart in HALCON/C++. Further restrictions can be found in section 4.1.4 on page 107.

Appendix A

Glossary

Boolean is the type name for the truth values `true` and `false` as well as for the related boolean expressions.

Body A body is part of a conditional instruction (`if`) or a loop (`while` or `for`) and consists of a sequence of operator calls. If you consider the `for`-loop, for instance, all operator calls, that are located between `for` and `endfor` form the body.

Button A button is part of a graphical user interface. With the mouse the user can press a button to cause an action to be performed.

Control data Control data can be either numbers (↑`integer` and ↑`real`), character strings (↑`string`) and truth values (`boolean`). This data can be used as atomic values (i.e., single values) or as ↑tuples (i.e., arrays of values).

Empty region An empty ↑region contains no points at all, i.e., its area is zero.

Graphics window A graphics window is used in ↑HDevelop for displaying ↑images, ↑regions, or ↑XLD.

HDevelop is an integrated development environment (IDE) for the creation of HALCON applications.

Iconic data are image data, i.e., image arrays and data, which are described by coordinates and are derived from image arrays, e.g., ↑regions, ↑image and ↑XLD.

Image An image consists of one or more (multichannel image) image arrays and a ↑region as the definition domain. All image arrays have the same dimension, but they can be of different pixel types. The size of the ↑region is smaller or equal than the size of the image arrays. The ↑region determines all image points that should be processed.

Iconic object Generic implementation of ↑iconic data in HALCON.

integer is the type name for integer numbers. Integers are implemented using the C-type `long` (4 or 8 byte).

Operator data base The operator data base contains information about the HALCON operators. They are loaded at runtime from the binary files in `%HALCONROOT%\help`.

Program window In HDevelop the program window contains the program. It is used to edit (copy, delete, and paste lines) and to run or debug the program.

Operator window In the operator window of HDevelop the parameters of the selected operators can be entered or modified.

Real is the type name for floating point numbers. They are implemented using the C-type `double` (8 bytes).

Region A region is a set of image points without gray values. A region can be imagined as a binary image (mask). Regions are implemented using runlength encoding. The region size is not limited to the image size (see also `set_system('clip_region','true'/'false')` in the HALCON reference manual.

String is the type name for character strings. A string starts and ends with a single quote; in between any character can be used except single quote. The empty string consists of two consecutive single quotes. The maximum length of a character string is limited to 1024 characters.

Tuple A tuple is an ordered multivalue set. In case of ↑control data a tuple can consist of a large number of items with different data types. The term tuple is also used in conjunction with ↑iconic objects, if it is to be emphasized that several ↑iconic objects will be used.

Type ↑iconic variables can be assigned with data items of type ↑image, ↑region, and ↑XLD. The types of ↑control data items can be one of ↑`integer`, ↑`real`, ↑`boolean`, or ↑`string`.

Variable window In HDevelop the variable window manages the ↑control and ↑iconic data.

XLD is the short term for e*X*tended *L*ine *D*escription. It is used as a superclass for contours, polygons, and lines (see also the HALCON Reference Manual).

Index